EWD ExtJS 4

Custom Tag Guide

Table of Contents

Introduction 1

Background 1

EWD ExtJS v4 Custom Tags: General Features 1

Container Pages and Fragments 1

ExtJS Custom Tags 2

Working from the ExtJS v4 Examples 6

ExtJS Custom Tag Definitions 8

Installation and Configuration 9

Installing the ExtJS v4 Tag Library 9

Configuring EWD for use with ExtJS v4 9

The <ext4:container> Tag 10

The <ext4:fragment> Tag 11

A Simple Hello World Application 12

Pre-requisites 12

Hello World version 1 12

Hello World Analysed 13

Creating your own Pointers to ExtJS Widgets 14

Generating the Hello World Panel Contents Dynamically 15

Nesting Panels 16

Adding Panels using Fragments 17

Using Javascript with ExtJS 18

Using Listeners 18

Using Explicitly-added Javascript 19

Requesting and Fetching Fragments 21

The EWD.ajax.getPage() Function 21

Preventing Unauthorised Use 22

The NextPage Attribute 23

ExtJS Layouts 25

Layouts 25

ExtJS Documentation Example 25

Layout Sub-components 28

Toolbars 30

Defining a Toolbar & Buttons 30

Formatting the Toolbar 32

Button Menus 33

Static Button Menus 33

Multi-level Static Menus 34

Adding Interactivity to Menu Items 34

Dynamically-defined Button Menus 35

Adding Interactivity to Dynamic Button Menus 35

Dynamically Defining Multi-level Button Menus 36

Identifying the Clicked Menu Item 36

Tools 37

Menus 39

Menu Types 39

Menus 39

Tree Menus 39

Combining a Tree Panel with a Layout 40

Tab Panels 44

Defining a Tab Panel 44

An Example 44

Adding Dynamic Behaviour 45

Windows 47

ExtJS Windows 47

Simple Example 47

Modal Windows 48

Grids 49

ExtJS Grids 49

Simple Example 49

Dynamic Columns 50

Identifying the Grid and Store 51

Special Column Types 52

Explicity-defined Example 52

Dynamically-defined Example 54

Editable Grids 56

The Editor Tag 56

Editing using the TextField Editor 56

Using Dynamically-defined Columns 58

Numberfield Editor 59

Datefield Editor 59

Combobox Editor 60

Grouping 62

Forms 63

ExtJS Forms 63

Text Fields 63

Date Fields 66

Time Fields 67

Number Fields 68

Display Fields 69

Hidden Fields 69

Slider Fields 70

Radio Fields 70

Checkbox Fields 71

Combobox Fields 72

MultiSelect Combobox Field 73

Textarea Fields 74

HTMLEditor Fields 74

The ExtJS Desktop 76

Demystifying the ExtJS Desktop 76

A Simple Desktop 76

The Logout Option 78

Defining a UserName 78

Adding a Login Mechanism 79

Customising & Extending the Desktop 80

Window Contents 80

Positioning Icons 80

Quickstart Icon 80

Desktop Icons 81

Window Icons 81

Desktop Wallpaper 81

Dynamically-defined Desktop 82

Appendix 1: Installing EWD 83

Installing EWD 83

GT.M 83

Caché 83

Configuring EWD 83

Caché & CSP 84

1a) Simple Default Configuration 84

1b) Custom Configuration 84

2) Define CSP Application 84

Caché & WebLink 85

GT.M 86

Creating EWD Pages 86

Running EWD Applications 87

Introduction

Background

Enterprise Web Developer (EWD) is a framework for rapidly building web applications that integrate with the Caché and GT.M databases. An important and powerful feature of EWD is its Custom Tags: these can be used to abstract and simplify the use of Javascript frameworks. This has two key benefits:

- reducing the learning curve of what are often complex and poorly-documented frameworks
- replacing programming code with a tag-based development framework. It is generally recognised that HTML and XML tags are a lot easier to read, understand and maintain than programming code.

This document describes one such EWD Custom Tag Library, which abstracts the ExtJS version 4 framework.

EWD ExtJS v4 Custom Tags: General Features

Container Pages and Fragments

The EWD ExtJS v4 Custom Tags introduce a new convention and shortcut to EWD: an abbreviated way to define Container Pages and Fragments.

> A *Container Page* is the initial complete page of HTML that is loaded into the browser and which remains in place throughout the user's session. *Fragments* are chunks of dynamically-generated markup that are injected into the container page (using AJAX/DOM techniques) and/or dynamically-generated Javascript or JSON code that is pulled into the Container Page's Javascript environment and automatically invoked.

You no longer have to add an *<ewd:config>* tag to the top of your pages and fragments, and you no longer need to specify any of the *<html>, <body>* and *<head>* markup tags in your container page.

Instead, you simply define a Container Page by using the *<ext4:container>* tag and a Fragment using the *<ext4:fragment>* tag. EWD's compiler will generate the appropriate *<ewd:config>* tag automatically for you, and will create all the necessary markup for your container pages.

When you're using the EWD ExtJS tags you will rarely want or need to specify any traditional HTML markup: everything you'll need to do will be using the new ExtJS v4 tags inside either *<ext4:container>* or *<ext4:fragment>* tags. Your EWD pages and fragments will now be even more succinct, readable and maintainable.

A typical example of a container page is shown below:

```
<ext4:container rootPath="/vista/ext-4">
  <ext4:window title="Resize Me" height="300" width="500" minWidth="300" minHeight="200" layout="fit"
plain="true" hidden="false">

  <ext4:formPanel border="false" bodyPadding="5">
    <ext4:fieldDefaults labelWidth="55" />
    <ext4:textfield fieldLabel="Send To" name="to" anchor="100%" />
    <ext4:textfield fieldLabel="Subject" name="subject" anchor="100%" />
    <ext4:textareafield hideLabel="true" name="msg" anchor="100% -47" />
  </ext4:formPanel>

   <ext4:toolbar dock="bottom" ui="footer">
     <ext4:fill />
     <ext4:button text="Send" />
     <ext4:button text="Cancel" />
   </ext4:toolbar>
  </ext4:window>
</ext4:container>
```

and a typical fragment:

```
<ext4:fragment onBeforeRender="getGridData^Ext4Demo">

 <ext4:gridPanel title="Simpsons" height="200" width="600" columnDefinition="colDef" sessionName="simpsons" stor-
eId="myStore" clicksToEdit="1" validationPage="gridValidateTest" />

</ext4:fragment>
```

Note the *onBeforeRender* attribute in the example above. This replaces the *prePageScript* attribute but is 100% analogous to it: the *onBeforeRender* attribute specifies the back-end Caché or GT.M method that is to be invoked before the contents of the container page or fragment is rendered and dispatched to the browser.

ExtJS Custom Tags

The EWD ExtJS v4 Custom Tags are designed to correspond directly to the ExtJS v4 API. For this reason, it is recommended that the developer makes use of the ExtJS v4 API documentation at:

http://docs.sencha.com/ext-js/4-0/#!/api

In general, every EWD ExtJS v4 Custom Tag represents either:

- a corresponding ExtJS class
- an ExtJS config option that is represented as an array of objects
- an object that corresponds to an ExtJS config option
- an ExtJS class that is embedded inside another, typically as one of its *item* objects

For example, the *<ext4:panel>* tag represents the ExtJS v4 *Ext.panel.Panel* class.

All the simple string (ie name/value pair) Config Options in an ExtJS class are represented as a correspondingly named attribute in the EWD Custom tag. For example:

```
<ext4:panel title="Hello" width="200" height="400" html="Hello World!" />
```

represents the following ExtJS v4 construct:

```
Ext.create('Ext.panel.Panel', {
    title: 'Hello',
    width: 200,
    height: 400,
    html: 'Hello World!'
});
```

Some of the ExtJS v4 class Config Options have values that are objects, arrays or arrays of objects. A number of these are very common and are used by many or most of the ExtJS classes. Examples include:

- items
- defaults
- dockedItems
- listeners
- layout

EWD provides a standard set of techniques for handling these.

defaults is an example of Config Option whose value is an object. For example:

```
Ext.create("Ext.container.Viewport", {
    layout: "border",
    defaults: {
        collapsible: true,
        split: true
    },
    ...etc
}
```

Any Config Option that has an object as a value is represented as its own EWD tag, and is nested inside the parent tag that represents the class. So, for example, the above construct would be represented, in EWD's ExtJS v4 tags as:

```
<ext4:viewPort layout="border">
  <ext4:defaults collapsible="true" split="true" />
</ext4:viewPort>
```

items, *tools*, *bbar*, *tbar* are examples of Config Options whose values are an array of objects. The general rule is that each of these can be represented as a corresponding tag (which takes no attributes) that represents the Config Option name, inside of which are child tags representing each object within the array, eg:

```
<ext4:panel title="My Window" height="450" width="600">
  <ext4:bbar>
    <ext4:item xtype="button" text="Button 1" />
    <ext4:item xtype="button" text="Button 2" />
  </ext4:bbar>
</ext4:panel>
```

In fact, in most cases, EWD also provides a higher-level child tag that automatically knows what Config Option tag it should be enclosed inside. So the following example which creates two panels, one nested inside the other:

```
<ext4:panel title="My Window" height="450" width="600">
  <ext4:items>
    <ext4:item xtype="panel" title="Panel 2" />
  </ext4:items>
</ext4:panel>
```

could also be represented more succinctly and more readably as:

```
<ext4:panel title="My Window" height="450" width="600">
    <ext4:panel title="Panel 2" />
</ext4:panel>
```

The inner *<ext4:panel>* tag knows to create itself as an *item* object with an *xtype* of "panel" inside an *items* Config Option. Both examples generate the same ExtJS Javascript:

```
Ext.create("Ext.panel.Panel", {
    height: 450,
    title: "My Window",
    width: 600,
    items: [{
        title: "Panel 2",
        xtype: "panel"
    }]
});
```

You can nest EWD ExtJS v4 tags to whatever depth you require, and the compiler will generate the correctly-structured code with *items* inside *items* to any depth.

As you can imagine, whilst deeply-nested ExtJS code can quickly become difficult to read and understand in terms of the widgets being used, the EWD abstraction into nested, intuitively-named XML tags retains readability (and hence maintainability).

Some of the EWD ExtJS v4 tags provide additional EWD-specific attributes that do not map to any ExtJS Config Options, but instead automate EWD-specific functionality such as integration with the

back-end Cache or GT.M database. For example the <ext4:gridPanel> tag not only represents the Ext.grid.Panel class, it also allows you to use attributes that allow you to define the grid contents and/or its column definitions in EWD Session Arrays. If these attributes are used, EWD will generate the grid dynamically at render time using the EWD Session Array contents.

Furthermore, the values of all attributes can be replaced with EWD Session values by using the syntax #, eg to use the value of an EWD Session value named *message*:

```
<ext4:panel html="#message" />
```

Working from the ExtJS v4 Examples

The EWD ExtJS v4 tags are designed so that any of the ExtJS v4 examples (http://docs.sencha.com/ext-js/4-0/#!/example) can be represented as a corresponding set of EWD tags. In most cases the key is understanding what the nested items represent. Where the xtype is explicitly defined in the example, this is not too difficult, but many examples make use of the implicitly-defined default xtype which makes it tricky to understand what's going on.

Sometimes the value of a nested widget will be separately defined as an object, and the name of/reference to that object will be used as the Config Option value instead.

For example, take a look at the Anchor Layout ExtJS example at http://docs.sencha.com/ext-js/4-0/#!/example/form/anchoring.html

If you click on the link for the Javascript source code, you'll see the core of it is as follows:

```
Ext.onReady(function() {
  var form = Ext.create('Ext.form.Panel', {
      border: false,
      fieldDefaults: {
          labelWidth: 55
      },
      url: 'save-form.php',
      defaultType: 'textfield',
      bodyPadding: 5,

      items: [{
          fieldLabel: 'Send To',
          name: 'to',
          anchor:'100%'  // anchor width by percentage
      },{
          fieldLabel: 'Subject',
          name: 'subject',
          anchor: '100%'  // anchor width by percentage
      }, {
          xtype: 'textarea',
          hideLabel: true,
          name: 'msg',
          anchor: '100% -47'  // anchor width by percentage and height by raw adjustment
      }]
  });

  var win = Ext.create('Ext.window.Window', {
      title: 'Resize Me',
      width: 500,
      height:300,
      minWidth: 300,
      minHeight: 200,
      layout: 'fit',
      plain: true,
      items: form,

      buttons: [{
          text: 'Send'
      },{
          text: 'Cancel'
      }]
  });

  win.show();
});
```

Like many of the official ExtJS examples, this example is rather confusing in many ways. It can actually be represented using EWD tags as follows:

```
  <ext4:window title="Resize Me" height="300" width="500" minWidth="300" minHeight="200" layout="fit"
plain="true" hidden="false">

  <ext4:formPanel border="false" bodyPadding="5">
    <ext4:fieldDefaults labelWidth="55" />
    <ext4:textfield fieldLabel="Send To" name="to" anchor="100%" />
    <ext4:textfield fieldLabel="Subject" name="subject" anchor="100%" />
    <ext4:textareafield hideLabel="true" name="msg" anchor="100% -47" />
  </ext4:formPanel>

   <ext4:buttons>
     <ext4:button text="Send" />
     <ext4:button text="Cancel" />
   </ext4:buttons>

  </ext4:window>
```

This is perhaps a surprisingly succinct abstraction of the ExtJS original, and it may not be immediately obvious how this abstraction is derived from the original example:

- by using *hidden="false"* in the *<ext4:window>* tag, we can avoid the need to explicitly invoke the window's *show()* method (*win.show()*)
- in the original example code, the *formPanel* is unnecessarily defined as a separate object and then referenced as the value of the window's *items* Config Option (*items: form*). In the EWD tag abstraction, we just need to nest an *<ext4:formPanel>* tag inside the *<ext4:window>* tag and it will automatically create the equivalent Javascript code. The EWD abstraction is immediately much more readable and understandable.
- in the original example code, the definition of the form fields inside the form panel is not very clear. In part this is because it relies on a *fieldDefaults* Config Option which specifies that each of the *item* objects inside the FormPanel's *items* Config Option is a *textfield*. In the EWD abstraction, we specifically define the field type explicitly via the tag name:
    - *ext4:textField*
    - *ext4:textAreaField*
- As a result, we've not needed to use an *<ext4:fieldDefaults>* tag, and it's much clearer what this example is going to do.

The *buttons* Config Option is actually a convenience alternative for the *dockedItems* Config Option, and we could rewrite our example as follows:

```
  <ext4:window title="Resize Me" height="300" width="500" minWidth="300" minHeight="200" layout="fit"
plain="true" hidden="false">

  <ext4:formPanel border="false" bodyPadding="5">
    <ext4:fieldDefaults labelWidth="55" />
    <ext4:textfield fieldLabel="Send To" name="to" anchor="100%" />
    <ext4:textfield fieldLabel="Subject" name="subject" anchor="100%" />
    <ext4:textareafield hideLabel="true" name="msg" anchor="100% -47" />
  </ext4:formPanel>

   <ext4:toolbar dock="bottom" ui="footer">
     <ext4:fill />
     <ext4:button text="Send" />
     <ext4:button text="Cancel" />
   </ext4:toolbar>

  </ext4:window>
```

This is functionally identical to the original use of the *<ext4:buttons>* and *<ext4:button>* tags. Which alternative you use is down to you: you need to decide which is the more intuitive, readable and maintainable alternative.

ExtJS Custom Tag Definitions

The EWD ExtJS Custom Tags also introduce a new and highly abstracted way of defining Custom Tags that are ultimately converted into Javascript. You can see the list of available tags and the JSON-based abstracted description of how they are processed by looking at the routine file *_zewdExt4Map.m* in the EWD GitHub Repository at:

https://github.com/robtweed/EWD

A description of the JSON abstracted tag definition conventions that have been used are beyond the scope of this current document, but advanced EWD developers should be aware that they can extend the tag set by adding further JSON-based definitions to the global:

- *^zewd("mappingObject","ext4")*

Installation and Configuration

Installing the ExtJS v4 Tag Library

The EWD ExtJS v4 Custom Tags are incorporated into EWD Build 912 and later. All you need to do is to install the ExtJS v4 Javascript library from the Sencha Site:

http://www.sencha.com/products/extjs/

Where you install ExtJS v4 is up to you, but it must be in a directory path that is accessible to your Web Server.

Configuring EWD for use with ExtJS v4

Configuration is very simple: in the *<ext4:container>* tag of your applications, you must specify the root web-server path that points to the directory path where you've installed the ExtJS v4 Javascript library, eg if you are using the dEWDrop VM (http://www.fourthwatchsoftware.com/) and you install ExtJS v4 in the physical path:

/home/vista/www/ext-4

then because the physical path */home/vista/www* is mapped in the Apache configuration file to the alias */vista*, the *<ext4:container>* tags in your EWD ExtJS v4 applications will look like this:

```
<ext4:container rootPath="/vista/ext-4">
  .... etc
</ext4:container>
```

The examples used in this document will all use the above rootPath. You should adjust your examples accordingly to reflect the location you've used for the ExtJS v4 Javascript files.

The <ext4:container> Tag

The first page of your ExtJS v4 applications should be defined inside an <ext4:container> tag. This tag will automatically create a standard EWD first page: during compilation it automatically generates an <ewd:config> tag as follows:

```
<ewd:config isFirstPage="true" cachePage="false">
```

The <ext4:container> tag has the following attributes:

Attribute	Purpose	Default Value
rootPath	Specifies the web-server alias path to the Sencha ExtJS v4 Javascript library files	/ext-4/
jsVersion	Optionally specifies the ExtJS Javascript file to load into the browser. Under most circumstances just accept the default, but this attribute allows you to optionally load the debug version, eg: jsVersion=”ext-all-debug.js”	“ext-all.js”
appName	Optionally allows you to specify an application name. This is used in the Ext.application() function call that is generated by EWD.	“Ext4App”
title	Optionally specified the value to be used in the container page’s generated <title> attribute	“ExtJS 4 Application”
enableLoader	Optionally allows you to enable the ExtJS library loader. If set to true, EWD will add the code: Ext.Loader.setConfig({enabled:true})	false
onBeforeRender	Optionally specifies a Caché Class Method or Extrinsic Function, or a GT.M Extrinsic Function that should be invoked prior to generating and rendering the container page contents. The onBeforeRender function is functionally identical to a prePageScript. It should have a single argument (sessid) and, if no error is to be automatically triggered, should QUIT with a null string as its returnValue.	Not applicable If not specified, no back-end function will

The <ext4:fragment> Tag

All fragments of your ExtJS v4 applications should be defined inside an <ext4:fragment> tag. During compilation, EWD automatically generates an <ewd:config> tag as follows:

```
<ewd:config isFirstPage="false" pageType="ajax">
```

The <ext4:fragment> tag has the following attributes:

Attribute	Purpose	Default Value
onBeforeRender	Optionally specifies a Caché Class Method or Extrinsic Function, or a GT.M Extrinsic Function that should be invoked prior to generating and rendering the fragment's contents. The onBeforeRender function is functionally identical to a prePageScript. It should have a single argument (sessid) and, if no error is to be automatically triggered, should QUIT with a null string as its returnValue.	Not applicable If not specified, no back-end function will

A Simple Hello World Application

Pre-requisites

In this chapter, we'll create a simple Hello World application, using an ExtJS Panel widget.

It is assumed that you've installed and configured EWD (See Appendix 1). Make sure you've installed Build 912 or later.

Hello World version 1

Create an EWD application directory named *ext4* and create a file named *helloworld1.ewd* that contains the following:

```
<ext4:container rootPath="/vista/ext-4">
 <ext4:panel title="Hello World Panel" html="Hello World" />
</ext4:container>
```

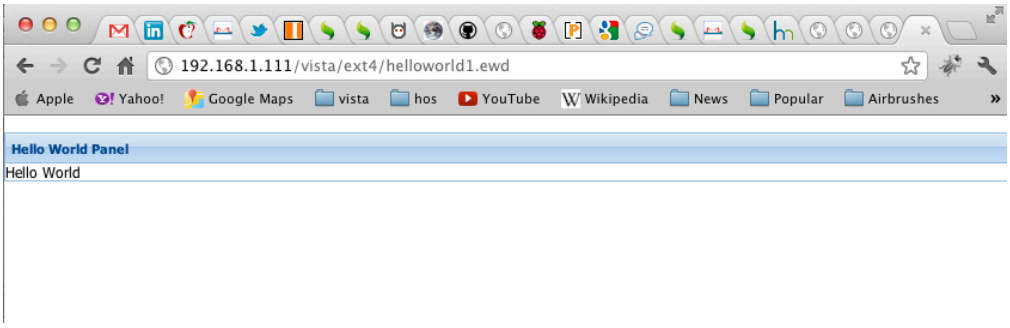
Save this file and compile it. For example, if you're running the GT.M-based dEWDrop VM, you'd type:

```
vista@dEWDrop:~$ mumps -dir

MU-beta>d compilePage^%zewdAPI("ext4","helloworld1")
/home/vista/www/ewd/ext4/ewdAjaxError.ewd
/home/vista/www/ewd/ext4/ewdAjaxErrorRedirect.ewd
/home/vista/www/ewd/ext4/ewdErrorRedirect.ewd
/home/vista/www/ewd/ext4/helloworld1.ewd

MU-beta>
```

Then start a browser, eg Chrome, and enter the appropriate URL to start it up. For example, if the dEWDrop VM is accessible at the IP address: 192.168.1.111, you'd use the URL:

http://192.168.1.111/vista/ext4/helloworld1.ewd

You should see something like the following:



If you see this, then your EWD and ExtJS v4 environment is working correctly. You've just created an ExtJS-based web application.

Hello World Analysed

What you've created is an ExtJS v4 Panel. Let's examine what you've created in some more detail. If you grab the source for the container page, you'll find that your three lines of EWD tags has been converted into a quite substantial page of HTML and Javascript. The critical part to examine is the main Javascript <script> tag that contains the following:

```
EWD.ext4 = {
    submit: function (formPanelId, nextPage, addTo, replace) {
        var nvp = '';
        var amp = '';
        Ext.getCmp(formPanelId).getForm().getFields().eachKey(

        function (key, item) {
            if ((item.xtype !== 'radiogroup') && (item.xtype !== 'checkboxgroup')) {
                var value = '';
                if (item.xtype === 'htmleditor') {
                    value = item.getValue();
                } else {
                    if (item.getSubmitValue() !== null) value = item.getSubmitValue();
                }
                nvp = nvp + amp + item.getName() + '=' + value;
                amp = '&';
            }
        });
        if (addTo !== '') nvp = nvp + '&ext4_addTo=' + addTo;
        if (replace === 1) nvp = nvp + '&ext4_removeAll=true';
        EWD.ajax.getPage({
            page: nextPage,
            nvp: nvp
        })
    }
};
Ext.application({
    name: 'ext4',
    launch: function () {
        EWD.ext4.content()
    }
});
EWD.ext4.content = function () {
    Ext.create("Ext.panel.Panel", {
        html: "Hello World",
        id: "panelhelloworld15",
        renderTo: Ext.getBody(),
        title: "Hello World Panel"
    });
}
```

You may find it useful to use the Online Javascript Beautifier (http://jsbeautifier.org/) to unpack and lay out the generated Javascript as shown above, to make it more readable.

Of the generated code shown above, the really important part is the following:

```
    Ext.create("Ext.panel.Panel", {
        html: "Hello World",
        id: "panelhelloworld15",
        renderTo: Ext.getBody(),
        title: "Hello World Panel"
    });
```

This is the ExtJS Javascript code that was specifically generated from your original <ext4:panel> tag. You'll see that your html and title attributes have been converted into corresponding object properties, and EWD's compiler has added two further attributes: *id* and *renderTo*. EWD assumed that you wanted the main panel rendered automatically in the document's <body> section, and added an id so that the panel widget could be uniquely identified using:

```
    Ext.getCmp("panelhelloworld15");
```

Creating your own Pointers to ExtJS Widgets

It is much more practical to provide our own id value, since we'll not easily be able to predict the id value that EWD will otherwise automatically assign. You can do this very simply by editing the EWD page as follows:

```
<ext4:container rootPath="/vista/ext-4">
 <ext4:panel title="Hello World Panel" html="Hello World" id="helloWorld" />
</ext4:container>
```

If you recompile this page, run it in the browser again and view the source, you'll see that the code has now changed to:

```
    Ext.create("Ext.panel.Panel", {
        html: "Hello World",
        id: "helloWorld",
        renderTo: Ext.getBody(),
        title: "Hello World Panel"
    });
```

Another way to obtain a pointer to the panel widget is to specify an object reference for the panel:

```
<ext4:container rootPath="/vista/ext-4">
 <ext4:panel title="Hello World Panel" html="Hello World" id="helloWorld"
object="myPanel" var="true" />
</ext4:container>
```

Compile, re-run and view the source, and you'll see that the generated code for the panel widget has now changed to:

```
    var myPanel = Ext.create("Ext.panel.Panel", {
        html: "Hello World",
        id: "helloWorld",
        renderTo: Ext.getBody(),
        title: "Hello World Panel"
    });
```

The attribute *var="true"* told the compiler to scope the object name (*myPanel*) using a *var* command. The panel can now be referenced either using the variable *myPanel* or the function *Ext.getCmp('helloworld').* If you omit the *var="true"* attribute or specify *var="false"*, then the object is declared as a global variable, ie:

```
    myPanel = Ext.create("Ext.panel.Panel", {
        html: "Hello World",
        id: "helloWorld",
        renderTo: Ext.getBody(),
        title: "Hello World Panel"
    });
```

It is generally recommended that in most circumstances, you should specify *var="true".*

Generating the Hello World Panel Contents Dynamically

Dynamic content is generated in EWD applications using the onBeforeRender function. Basically it's a two-step approach:

- write a Cache or GT.M function that creates and/or gets data (eg from the database)

- save that data into the EWD Session as simple variables or multi-dimensional arrays

The EWD tags can then make use of data in the EWD Session. Some of the EWD ExtJS v4 tags automatically know how to use data stored in EWD Session Arrays, but simple EWD Session Variables can be used as the value of any ExtJS tag attribute. The latter capability is very simple: instead of using a quoted literal value, just add a **#** symbol at the start of the value, eg:

```
<ext4:container rootPath="/vista/ext-4">
 <ext4:panel title="Hello World Panel" html="#hello" id="helloWorld"
object="myPanel" var="true" />
</ext4:container>
```

In the example above, we've substituted the literal value for the *html* attribute with a reference to an EWD Session Variable named ***hello***. In effect, what this is saying is: the value of the *html* attribute will be whatever is currently held in an EWD Session Variable named *hello*.

What's missing from the example above is the means by which the EWD Session variable named hello is created. We therefore need to add an onBeforeRender attribute to the <ext4:container> tag, eg:

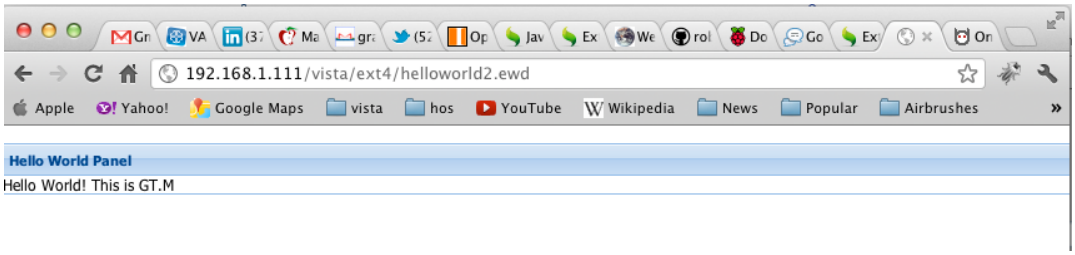
```
<ext4:container rootPath="/vista/ext-4" onBeforeRender="getHello^ext4Demo" >
 <ext4:panel title="Hello World Panel" html="#hello" id="helloWorld"
object="myPanel" var="true" />
</ext4:container>
```

In the example above, a GT.M function named *getHello()* from a routine file named *ext4Demo* will be invoked before the Container Page is generated and rendered. This function could look something like this:

```
getHello(sessid) ;
 d setSessionValue^%zewdAPI("hello","Hello World! This is GT.M",sessid)
 QUIT ""
```

If you're using the dEWDrop VM, you can create this using a text editor. Save it as */home/vista/p/ext4Demo.m*

Recompile the new version of helloworld1.ewd and re-run the URL again. It should now look like this:



There's the text we defined in the onBeforeRender method!

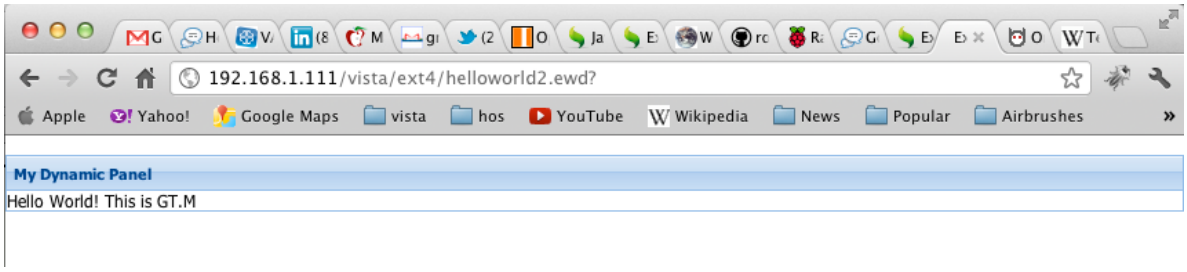
In the getHello() function above, the value is being defined as a string literal, but of course it could have been the result of a database query.

We could make the panel title dynamic also:

```
<ext4:container rootPath="/vista/ext-4" onBeforeRender="getHello^ext4Demo" >
 <ext4:panel title="#myTitle" html="#hello" id="helloWorld" object="myPanel"
var="true" />
</ext4:container>
```

```
getHello(sessid) ;
 d setSessionValue^%zewdAPI("hello","Hello World! This is GT.M",sessid)
 d setSessionValue^%zewdAPI("myTitle","My Dynamic Panel",sessid)
 QUIT ""
```

Save the EWD page and routine files as amended above and recompile the EWD page, and you should now see:



Nesting Panels

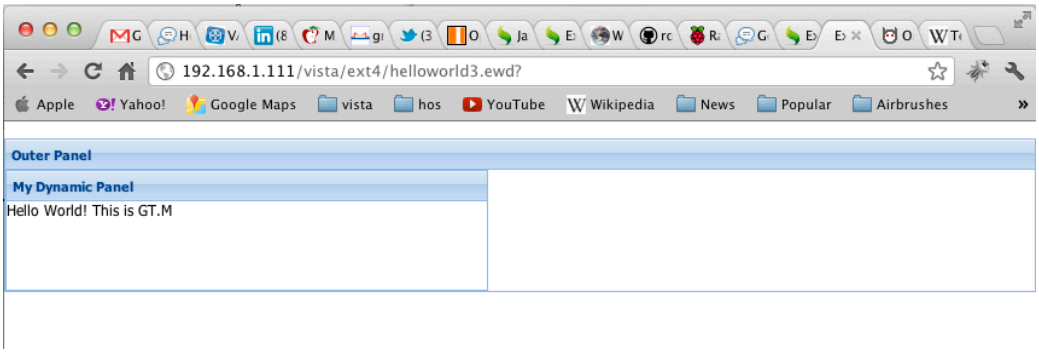
Although you can mix standard HTML markup with ExtJS widgets, ExtJS is really designed to be self-contained and most UI effects you'll need can be achieved using ExtJS widgets and their APIs.

The Panel is probably the most basic UI unit in ExtJS: you'll find that you'll be using them all over the place in many different ways. Panels can be nested inside each other and alongside others. So let's extend our simple Hello World application to demonstrate how this can be done.

EWD makes it extremely simple and intuitive, eg:

```
<ext4:container rootPath="/vista/ext-4" onBeforeRender="getHello^ext4Demo" >
 <ext4:panel title="Outer Panel" id="helloWorld1" object="myPanel" var="true">
  <ext4:panel title="#myTitle" html="#hello" id="helloWorld2" width="400"
height="100" />
 </ext4:panel>
</ext4:container>
```

This creates the following:



Adding Panels using Fragments

ExtJS allows you to achieve the same effect dynamically using Javascript. Panels expose an add() method that can be used to dynamically add one panel to another. EWD's ExtJS v4 tags automatically use this mechanism when you use Fragments to inject new dynamically-generated widgets into what's already being displayed in your Container Page.

So, for example, we could rewrite the previous example using a Container Page and Fragment as follows:

Container Page: helloworld4.ewd

```
<ext4:container rootPath="/vista/ext-4">
 <ext4:panel title="Outer Panel 2" id="helloWorld1" object="myPanel" var="true"
addPage="hwFragment4a" />
</ext4:container>
```

Fragment: hwFragment4a.ewd

```
<ext4:fragment onBeforeRender="getHello^ext4Demo">
  <ext4:panel title="Inner Injected Panel" html="#hello" id="helloWorld2"
width="400" height="100" />
</ext4:fragment>
```

The key attribute is *addPage*. This does two things:

- asks EWD to fetch the fragment named *hwFragment4a.ewd*

- after this fragment is rendered, the *add()* method is invoked for the panel owning the *addPage* attribute. The panel that is added is the outer one in the fragment being fetched.

So, if the fragment contained a pair of nested panels, they'll be added to the Container Page's panel, eg see what happens if you edit hwFragment4a.ewd as follows:

```
<ext4:fragment onBeforeRender="getHello^ext4Demo">
  <ext4:panel title="Outer Injected Panel" id="helloWorld2" width="400"
height="100">
    <ext4:panel title="Inner Injected Panel" html="#hello" />
  </ext4:panel>
</ext4:fragment>
```

The *addPage* attribute can be used with most ExtJS component widgets. By default, the fragment that is fetched will be added along with any other panels (or other component widgets) that have been previously added. However, if you also include the attribute *replacePreviousPage="true"*, then the fetched fragment will replace any previously added fragment panels (or other component widgets).

Using Javascript with ExtJS

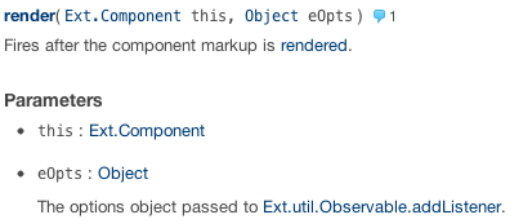
There are two ways in which you can use Javascript in conjunction with EWD's ExtJS tags:

- using widget-specific listeners

- using explicitly added Javascript code within Fragments

Using Listeners

Listeners should be used whenever possible and applicable: they provide the most elegant approach. The events that are available for each ExtJS widget or component is fully documented. To add a listener, just add an *<ext4:listeners>* tag inside the ExtJS widget tag, with one or more event-specific *<ext:listener>* tag inside. For example, panels include a *render* event that will fire when the panel is rendered. You'll find it documented under the Events section of the API documentation for Ext.panel.Panel as follows:



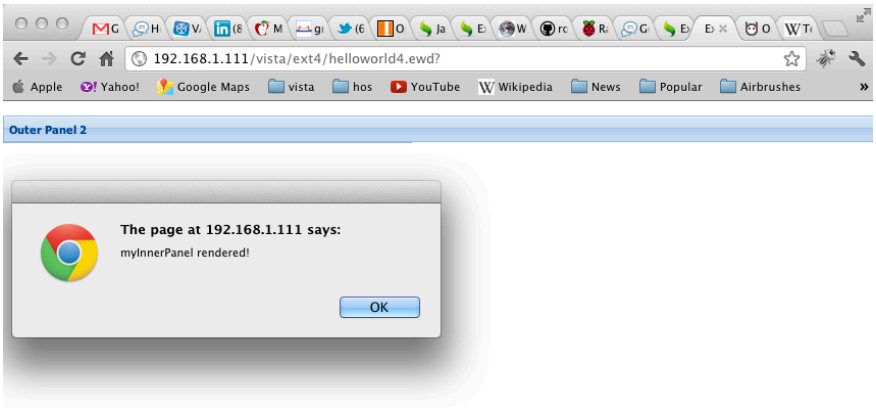
To make use of this in our example, we could edit the *hwFragment4a.ewd* fragment as follows:

```
<ext4:fragment onBeforeRender="getHello^ext4Demo">
  <ext4:panel title="Outer Injected Panel" id="helloWorld2" width="400"
height="100">
    <ext4:panel title="Inner Injected Panel" html="#hello">
      <ext4:listeners>
        <ext4:listener render="alert('Rendered!')" />
      </ext4:listeners>
    </ext4:panel>
  </ext4:panel>
</ext4:fragment>
```

Most listeners have parameters that may be used by your function. To make use of these, you need to replace the abbreviated render function that we've used in the example above with a full function specification. The documentation for the render event shows it has two parameters: ***this*** and ***eOpts***. *this* provides a pointer to the panel that owns the listener. So, we could enhance the example above as follows:

```
<ext4:fragment onBeforeRender="getHello^ext4Demo">
  <ext4:panel title="Outer Injected Panel" id="helloWorld2" width="400"
height="100">
    <ext4:panel title="Inner Injected Panel" id="myInnerPanel" html="#hello">
      <ext4:listeners>
        <ext4:listener render="function(panel,eopts) {alert(panel.id + ' ren-
dered!');}" />
      </ext4:listeners>
    </ext4:panel>
  </ext4:panel>
</ext4:fragment>
```

Now our render listener has access to the panel object, so we can display its *id* property which we've defined to be the literal text *"myInnerPanel"*. So when the example above is run, we'll see the following alert appear:



Note: if your listener logic is lengthy, it is recommended that you define it as a separate function in a static *.js* file, and invoke that function from within the *<ext4:listener>* tag. Long attribute values containing Javascript logic can quickly reduce the readability (and hence the maintainability) of EWD pages or fragments For example the alert logic could be defined separately in a function named *alertMe()*, allowing us to reduce the render attribute to the following:

```
<ext4:listener render="function(panel,eopts) {alertMe(panel);}" />
```

The *<ext4:listeners>* tag is available for use with any ExtJS tag that maps to a corresponding ExtJS component widget for which documented Events exist.

Using Explicitly-added Javascript

In some situations, it's more appropriate to add explicit blocks of Javascript code to your Fragments. You can specify whether your block of Javascript code is to be added before or after any generated ExtJS Javascript code.

Note: to ensure maximum readability and maintainability of your pages and fragments, you should endeavour to use static Javascript, defined in static .js files, whenever possible. In-line Javascript blocks within your fragments should be restricted

to logic that has to be dynamically defined at run-time, providing, for example, a specific and probably different value each time a fragment is rendered.

Static Javascript files can be requested and loaded into your Container page using a standard <script> tag, eg:

```
<ext4:container rootPath="/vista/ext-4">
 <script src="/vista/js/ext4Demo.js" />
 <ext4:panel title="Outer Panel 2" id="helloWorld1" object="myPanel" var="true"
addPage="hwFragment4a" />
</ext4:container>
```

You can defer execution of the script file by adding the attribute *defer="defer"*.

You can also add in-line Javascript to your Container Page, though this should be kept to a minimum, eg:

```
<ext4:container rootPath="/vista/ext-4">
 <script src="/vista/js/ext4Demo.js" />
 <ext4:panel title="Outer Panel 2" id="helloWorld1" object="myPanel" var="true"
addPage="hwFragment4a" />
 <script type="text/javascript">
   alert("in line Javascript here!");
 </script>
</ext4:container>
```

Inline Javascript within Fragments has to be handled differently: you embed any Javascript code inside *<ext4:js>* tags. The *<ext4:js>* tag has one mandatory attribute: ***at***. Possible values are *"top"* and *"bottom"*, denoting whether the inline Javascript is to be placed before or after any generated ExtJS Javascript. For example:

```
<ext4:fragment onBeforeRender="getHello^ext4Demo">
  <ext4:panel title="Outer Injected Panel" id="helloWorld2" width="400"
height="100">
    <ext4:panel title="Inner Injected Panel" id="myInnerPanel" html="#hello">
      <ext4:listeners>
        <ext4:listener render="function(panel,eopts) {alert(panel.id + ' ren-
dered!');}" />
      </ext4:listeners>
    </ext4:panel>
  </ext4:panel>

  <ext4:js at="top">
    alert("This goes before any ExtJS code");
  </ext4:js>

  <ext4:js at="bottom">
    alert("This goes after any ExtJS code");
  </ext4:js>

</ext4:fragment>
```

The generated Javascript that is sent to the browser when the example fragment above is loaded is as follows (after indentation by the Online Javascript Beautifier):

```
alert("This goes before any ExtJS code");
Ext.create("Ext.panel.Panel", {
    height: 100,
    id: "helloWorld2",
    title: "Outer Injected Panel",
    width: 400,
    items: [{
        html: "Hello World! This is GT.M",
        id: "myInnerPanel",
        title: "Inner Injected Panel",
        xtype: "panel",
        listeners: {
            render: function (panel, opts) {
                alertMe(panel);
            }
        }
    }]
});
var addTo = 'helloWorld1';
var remove = '';
if (remove === 'true') Ext.getCmp('helloWorld1').removeAll(true);
if (addTo !== '') Ext.getCmp('helloWorld1').add(Ext.getCmp('helloWorld2'));
alert("This goes after any ExtJS code");
```

You can see how the two alerts have been added before and after the rest of the code that was generated from the *<ext4:panel>* tag and its child tags.

Requesting and Fetching Fragments

The EWD ExtJS implementation provides you with a variety of ways by which you can request and fetch Fragments. You've seen one already: the *addPage* attribute. You can also use:

- the *EWD.ajax.getPage()* function

- the nextPage attribute for buttons and tree menu members

The EWD.ajax.getPage() Function

You can use the *EWD.ajax.getPage()* function anywhere within listeners or inline Javascript. The arguments for this function are as follows:

```
EWD.ajax.getPage({
  page: 'myFragment',
  targetId: 'myTarget',
  nvp: 'a=123&b=xyz'
});
```

Note:

- the *targetId* parameter is only required for fragments that deliver HTML markup. Leave it out if the fragment being fetched consists entirely of *<ext4:*>* tags and/or Javascript

- the *nvp* parameter is optional. It allows you to add additional name/value pairs to the HTTP request. These can be accessed and used within the Fragment's *onBeforeRender* method by using the *getRequestValue()* function, eg:

    - *set a=$$getRequestValue^%zewdAPI("a",sessid)*

Preventing Unauthorised Use

All fragments can, in theory, be fetched by a valid user by using the E*WD.ajax.getPage()* function. You should therefore always protect critical fragments from unauthorised access by someone using a development tool such as Chrome Developer Tools: such tools can allow arbitrary invocation of Javascript within the browser. EWD provides a mechanism whereby you can take fine-grained and, if required, context-sensitive control of which fragments may and may not be accessed at any point in the user's session. ***It is strongly recommended that you make use of this mechanism to prevent unauthorised access to fragments by Javascript hacking.***

The key to securing and controlling access to fragments is to add the *disableGetPage="true"* attribute to your *<ext4:container>* tags, for example:

```
<ext4:container rootPath="/vista/ext-4" disableGetPage="true">
 <ext4:panel title="Outer Panel 2" id="helloWorld1" object="myPanel" var="true"
addPage="hwFragment4a" />
</ext4:container>
```

Note: this attribute can also be used in older Container Pages that use the *<ewd:config>* tag.

The effect of adding this attribute is to prevent any fragments from being fetched. Clearly this may be too draconian and you may want the container page, itself, to be able to fetch one or more specific fragments (eg the example above needs to be able to fetch *hwFragment4a.ewd*). This can be achieved by adding an *onBeforeRender* method to the Container Page (or a pre-page script in an older-style Container Page) that makes use of the *enableGetPage()* EWD API method. For example:

```
<ext4:container rootPath="/vista/ext-4" disableGetPage="true"
   onBeforeRender="initAccess^Ext4Demo">
 <script src="/vista/js/ext4Demo.js" />
 <ext4:panel title="Outer Panel 2" id="helloWorld1" object="myPanel" var="true"
addPage="hwFragment4a" />
</ext4:container>
```

In the example above, the *initAccess()* function would release access to the *hwFragment4a.ewd* fragment as follows:

```
initAccess(sessid)
 d enableGetPage^%zewdAPI("hwFragment4a",sessid)
 QUIT ""
```

An alternative technique is also possible whereby you don't use the *disableGetPage* attribute, leaving the Container Page to adopt its default behaviour, allowing **all** fragments to be accessed. In the *onBeforeRender* method, you can then specify the fragment(s) to which you want to specifically deny access. For this you use the *disableGetPage()* EWD API, eg:

```
denyAccess(sessid)
 d disableGetPage^%zewdAPI("privilegedInfo",sessid)
 QUIT ""
```

You can probably see that by using the *enableGetPage()* and *disableGetPage()* APIs within your fragments' *onBeforeRender* methods, you can apply very fine-grained and context-sensitive access to the fragments available within an application, with each fragment selectively turning on and off access to other fragments (or even itself).

Note that the access control to fragments is determined at the back-end, out of sight from and inaccessible by the user. This is critically important: even though the initial Container Page creates Javascript functions that can potentially fetch every

one of the available fragments in the application, they will simply return an error if the back-end has been instructed to deny access. A hacker will only be able to fetch the fragments that the application would allow the user to fetch anyway at that particular point in their session.

The NextPage Attribute

The <ext4:button> tag allows the use of an attribute named *nextPage*. This attribute creates a handler function with a request to fetch the specified fragment, so it provides a very convenient and readable shorthand describing the functionality of the button. For example:

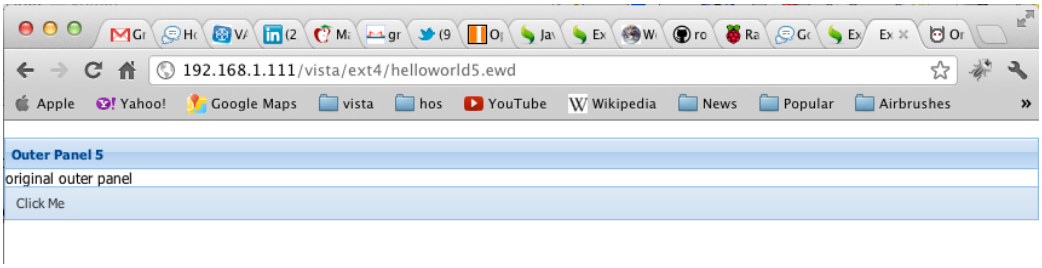
```
<ext4:container rootPath="/vista/ext-4">
 <ext4:panel title="Outer Panel 5" id="helloWorld1" html="original outer panel">
   <ext4:toolbar dock="bottom">
     <ext4:button text="Click Me" nextpage="hwFragment5b" addTo="helloWorld1" />
   </ext4:toolbar>
 </ext4:panel>
</ext4:container>
```

Clicking the button will fetch a fragment named *hwFragment5b.ewd*. The *addTo* attribute specifies the *id* of the ExtJS component to which the fragment's contents should be added. In this case we want to add it to the outer panel: its *id* is *"helloWorld1"*.

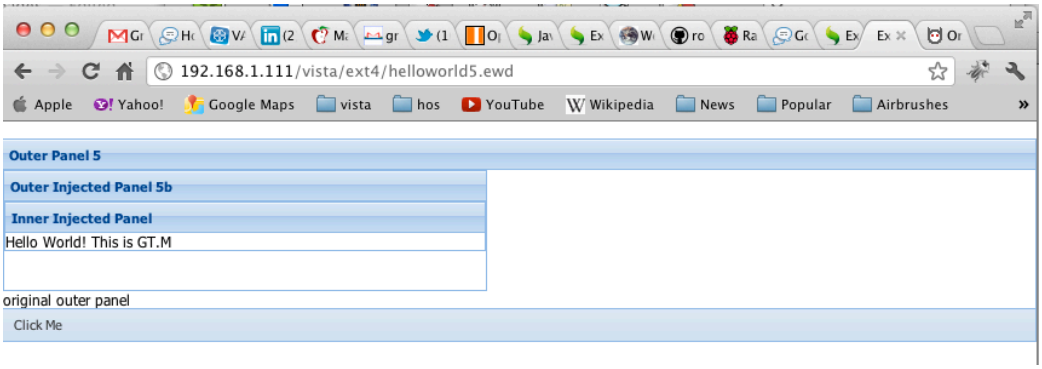
If *hwFragment5b.ewd* contains the following:

```
<ext4:fragment onBeforeRender="getHello^ext4Demo">
  <ext4:panel title="Outer Injected Panel 5b" id="panel5b1" width="400" height="100">
    <ext4:panel title="Inner Injected Panel" id="panel5b2" html="#hello" />
  </ext4:panel>
</ext4:fragment>
```

Then when the Container page is loaded, it will appear as follows:



When the Click Me button is clicked, the fragment is fetched and the Container Page changes to the following:



Note: you can optionally add another attribute named *nvp*: this allows you to add an additional name/value pair (or list of name/value pairs) to the generated request. This can help in your back-end code when you want to uniquely identify the button that requested the fragment, eg:

```
<ext4:button text="Click Me" nextpage="hwFragment5b" addTo="helloWorld1"
   nvp="a=12&b=xyz" />
```

The value of nvp could, of course, be an EWD Session variable:

```
<ext4:button text="Click Me" nextpage="hwFragment5b" addTo="helloWorld1"
   nvp="#but1nvp" />
```

Of course, you are not limited to this behaviour for buttons. If you require other behaviour, you can define a handler using the standard, documented handler Config Option, eg:

```
<ext4:button text="Try Me" handler="function() {alert('You clicked the button')}" />
```

ExtJS Layouts

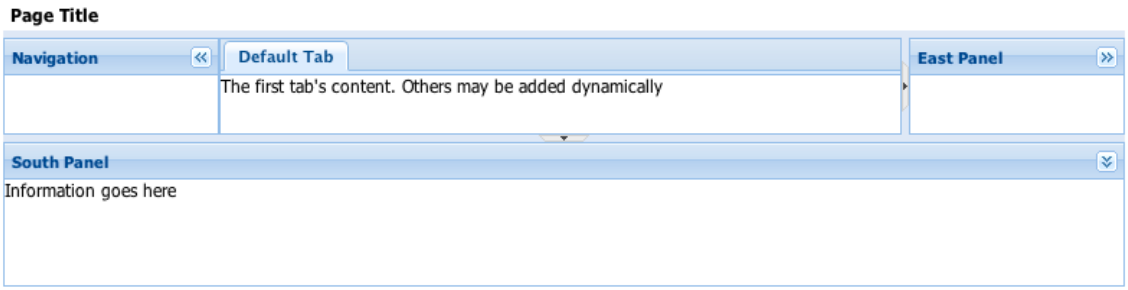
Layouts

ExtJS provides a very sophisticated and visually rich set of components and widgets for laying out the information that you display in a browser. The key components and their corresponding EWD Custom Tags include:

ExtJS Component	EWD Custom Tag	Purpose
Ext.container.Viewport	ext4:viewport	Creates a container that occupies the available browser area, into which you add panels, typically using regions
Standard Ext.panel.Panel	ext4:panel	Creates a panel container. Specify a region (eg North, South, Center, etc) to position it within a Viewport
Ext.panel.Panel defined to act as an accordion panel	ext4:accordionPanel	Creates an accordion panel
Ext.tab.Panel	ext4:tabPanel	Creates a tab panel that may contain a set of panels. Clicking a tab brings its associated panel into view

ExtJS Documentation Example

If you look at Sencha's API documentation for the Ext.container.Viewport, you'll see the following example:



The ExtJS Source Javascript code for this is as follows:

```
Ext.create('Ext.container.Viewport', {
    layout: 'border',
    items: [{
        region: 'north',
        html: '<h1 class="x-panel-header">Page Title</h1>',
        autoHeight: true,
        border: false,
        margins: '0 0 5 0'
    }, {
        region: 'west',
        collapsible: true,
        title: 'Navigation',
        width: 150
        // could use a TreePanel or AccordionLayout for navigational items
    }, {
        region: 'south',
        title: 'South Panel',
        collapsible: true,
        html: 'Information goes here',
        split: true,
        height: 100,
        minHeight: 100
    }, {
        region: 'east',
        title: 'East Panel',
        collapsible: true,
        split: true,
        width: 150
    }, {
        region: 'center',
        xtype: 'tabpanel', // TabPanel itself has no title
        activeTab: 0,      // First tab active by default
        items: {
            title: 'Default Tab',
            html: 'The first tab\'s content. Others may be added dynamically'
        }
    }]
});
```

This same example can be defined as follows using EWD:

```
<ext4:container rootPath="/ext-4">
  <ext4:viewPort layout="border">
    <ext4:panel region="north" id="northPanel" autoheight="true" border="false" margins="0 0 5 0"
      html="<h1 class='x-panel-header'>Page Title</h1>" />
    <ext4:panel region="west" collapsible="true" title="Navigation" width="150" />
    <ext4:panel region="south" collapsible="true" title="South Panel" collapsible="true"
     html="Information goes here" split="true" height="100" minHeight="100" />
    <ext4:panel region="east" collapsible="true" title="East Panel" split="true" width="150" />
    <ext4:tabPanel region="center" activeTab="0">
      <ext4:panel title="Default Tab"
         html="The first tab's content. Others may be added dynamically" />
    </ext4:tabPanel>
  </ext4:viewPort>
</ext4:container>
```

Of course, where we've defined HTML content for panels using the *html* attribute above, we could load the content dynamically using fragments, loaded using the *addPage* attribute. For example here's the same example rewritten using fragments:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getLayoutTitle^ext4Demo">
  <ext4:viewPort layout="border">
    <ext4:panel region="north" id="northPanel" autoheight="true" border="false" margins="0 0 5 0"
       html="<h1 class='x-panel-header'><?= #pageTitle ?></h1>" />
    <ext4:panel region="west" collapsible="true" title="Navigation" width="150" />
    <ext4:panel region="south" collapsible="true" title="South Panel" collapsible="true"
      split="true" height="100" minHeight="100" addPage="test1cSouthPanel" />
    <ext4:panel region="east" collapsible="true" title="East Panel" split="true" width="150" />
    <ext4:tabPanel region="center" activeTab="0">
      <ext4:panel title="Default Tab" addPage="test1cTabPanel" />
    </ext4:tabPanel>
  </ext4:viewPort>
</ext4:container>
```

The *onBeforeRender* function might be something like this:

```
getLayoutTitle(sessid)
 d setSessionValue^%zewdAPI("pageTitle","Dynamic Page Title",sessid)
 QUIT ""
```

The *test1cSouthPanel.ewd* fragment might be as follows:

```
<ext4:fragment onBeforeRender="getSouthPanelContent^ext4Demo">
 <ext4:panel border="0" html="#southPanelContent" />
</ext4:fragment>
```

And the *test1cTabPanel.ewd* fragment might be as follows:

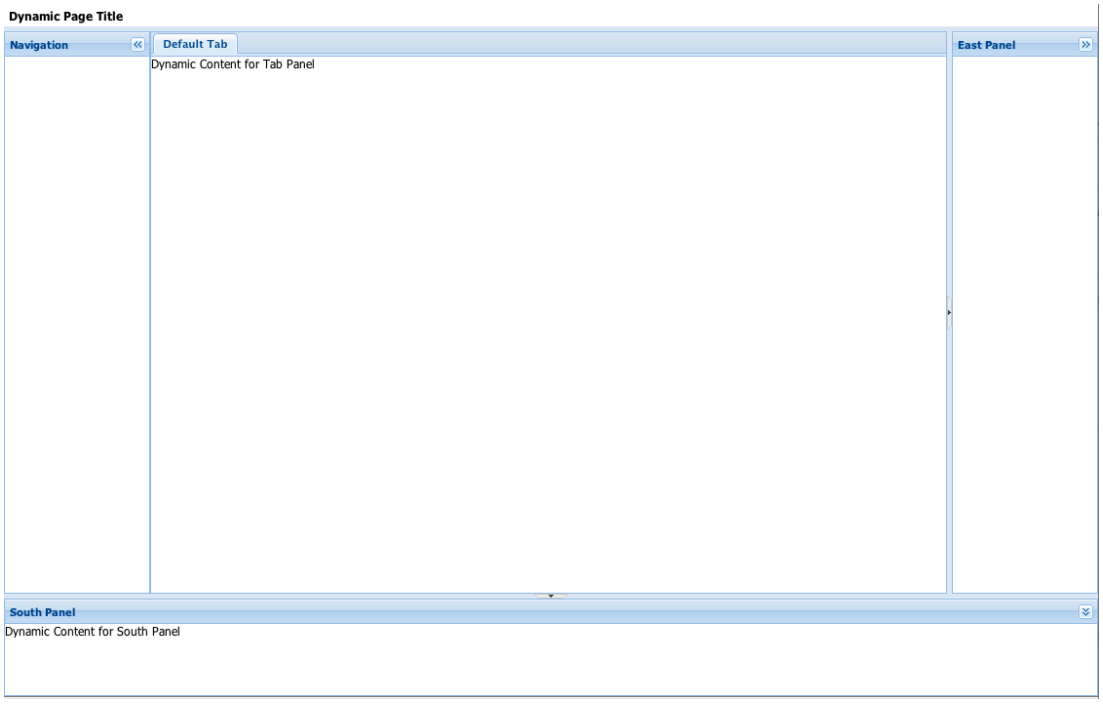
```
<ext4:fragment onBeforeRender="getTabPanelContent^ext4Demo">
 <ext4:panel border="0" html="#tabPanelContent" />
</ext4:fragment>
```

The onBeforeRender functions for these two fragments might be as follows:

```
getSouthPanelContent(sessid)
 d setSessionValue^%zewdAPI("southPanelContent","Dynamic Content for South Panel",sessid)
 QUIT ""
 ;
getTabPanelContent(sessid)
 d setSessionValue^%zewdAPI("tabPanelContent","Dynamic Content for Tab Panel",sessid)
 QUIT ""
```

Of course, in a real application, the dynamically-generated content would probably include data looked up from the GT.M or Caché database.

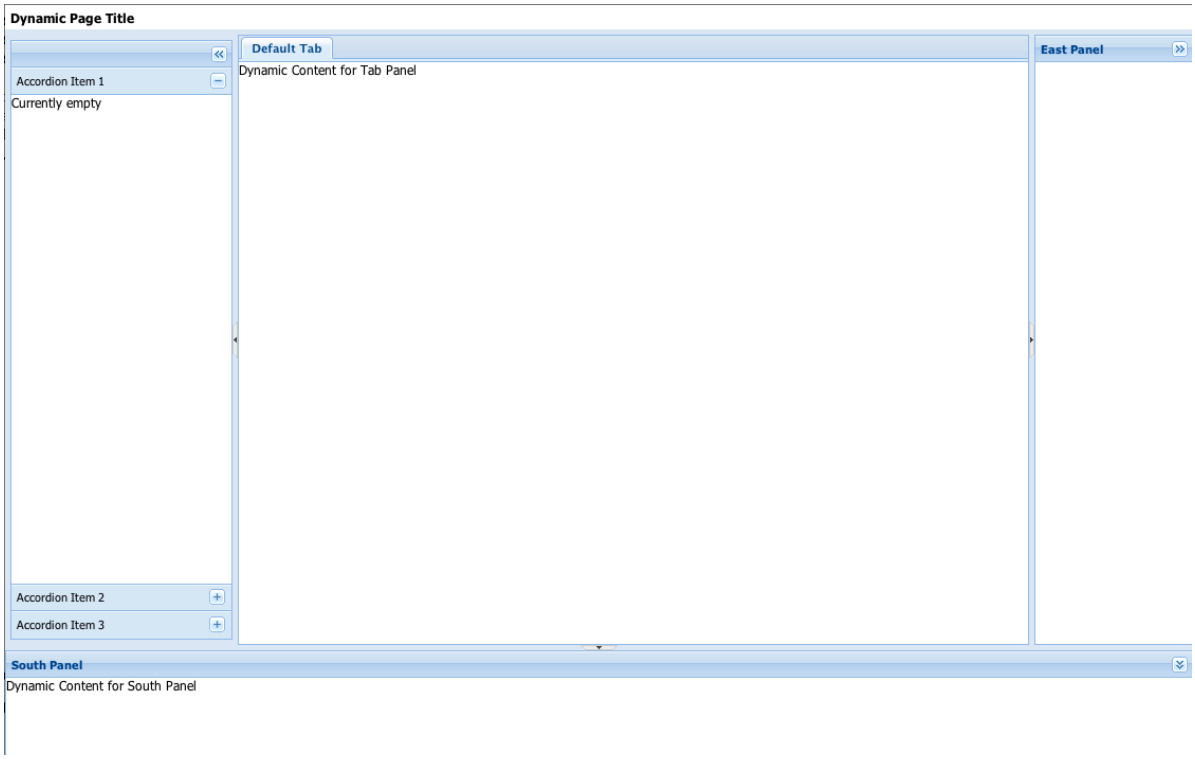
It will now look as follows when run in a browser:



It's very simple to replace the simple Navigation panel with a set of Accordion Panels:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getLayoutTitle^ext4Demo">
  <ext4:viewPort layout="border">
    <ext4:panel region="north" id="northPanel" autoheight="true" border="false" margins="0 0 5 0"
       html="<h1 class='x-panel-header'><?= #pageTitle ?></h1>" />
    <ext4:accordionPanel region="west" margins="5 0 5 5" split="true" collapsible="true"
     width="210">
      <ext4:panel title="Accordion Item 1" html="Currently empty" />
      <ext4:panel title="Accordion Item 2" html="Currently empty" />
      <ext4:panel title="Accordion Item 3" html="Currently empty" />
    </ext4:accordionPanel>
    <ext4:panel region="south" collapsible="true" title="South Panel" collapsible="true"
      split="true" height="100" minHeight="100" addPage="test1cSouthPanel" />
    <ext4:panel region="east" collapsible="true" title="East Panel" split="true" width="150" />
    <ext4:tabPanel region="center" activeTab="0">
      <ext4:panel title="Default Tab" addPage="test1cTabPanel" />
    </ext4:tabPanel>
  </ext4:viewPort>
</ext4:container>
```

The layout now appears as follows. Try it out for yourself and see how you can expand and contract each of the accordion panels.



Each of the panels within the *<ext4:accordionPanel>* container could, of course, be given dynamic content by replacing the *html* attribute with an *addPage* attribute that fetches the content from a fragment.

Layout Sub-components

Of course once you have a layout defined, you'll want to populate the various panels and containers with content, often included in and/or presented using other ExtJS widgets and components, for example:

- grids

- tree menus

- toolbars and panels

All of these components can also be used separately or in combination with other components: ie they aren't limited to use within a Viewport container.

The following chapters will focus on the most important of these ExtJS components, and how to use them with the EWD ExtJS v4 Custom Tags.

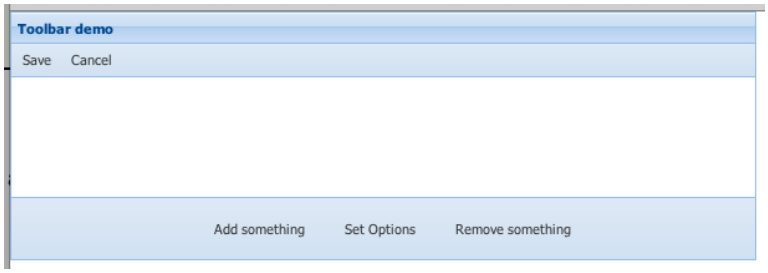
Toolbars

Defining a Toolbar & Buttons

Toolbars are header or footer bars that can be added to a Panel component, into which you can add buttons. The buttons can be configured to trigger actions such as requesting new EWD fragments. Let's look at a simple EWD example demonstrating their use:

```
<ext4:container rootPath="/ext-4">

  <ext4:panel width="600" height="200" title="Toolbar demo">

    <ext4:toolbar dock="top">
      <ext4:button text="Save" />
      <ext4:button text="Cancel" />
    </ext4:toolbar>

    <ext4:toolbar dock="bottom" height="50">
      <ext4:layout pack="center" />
      <ext4:button text="Add something" />
      <ext4:button text="Set Options" />
      <ext4:button text="Remove something" />
    </ext4:toolbar>

  </ext4:panel>

</ext4:container>
```

If you compile and run this page, it will appear as follows in your browser:



This example demonstrates how toolbars can be added to either the header or footer of a panel, and can either be left-justified or centered. Currently none of the buttons do anything. We could add a simple handler to the Save button, generating an alert if pressed:

```
<ext4:toolbar dock="top">
  <ext4:button text="Save" handler="function() {Ext.Msg.alert('Test','You clicked the Save button')}" />
  <ext4:button text="Cancel" />
</ext4:toolbar>
```

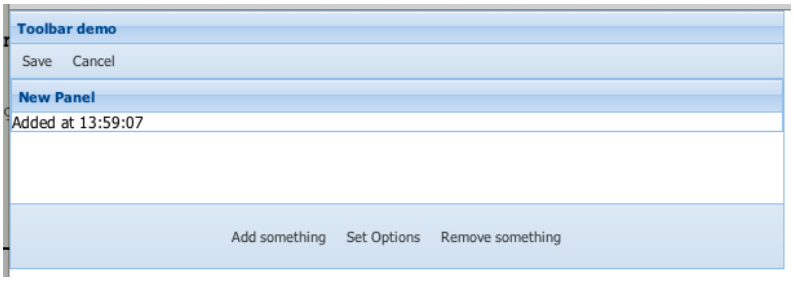
However, typically we'll want to fetch an EWD fragment, so we could do the following:

```
<ext4:container rootPath="/ext-4">

  <ext4:panel id="mainPanel" width="600" height="200" title="Toolbar demo">

    <ext4:toolbar dock="top">
      <ext4:button text="Save"
         handler="function() {Ext.Msg.alert('Test','You clicked the Save button')}" />
      <ext4:button text="Cancel" />
    </ext4:toolbar>

    <ext4:toolbar dock="bottom" height="50">
      <ext4:layout pack="center" />
      <ext4:button text="Add something" nextPage="test2cPanel" addTo="mainPanel"
         replacePreviousPage="true" />
      <ext4:button text="Set Options" />
      <ext4:button text="Remove something" />
    </ext4:toolbar>

  </ext4:panel>

</ext4:container>
```

If *testcPanel.ewd* contains the following:

```
<ext4:fragment>
 <ext4:panel title="New Panel" html="Added at <?= #ewd.time ?>" />
</ext4:fragment>
```

Then when you click the *Add something* button, you'll see something like:



By specifying *replacePreviousPage="true"*, the inner panel will be replaced each time you press the *Add something* button and will show the current time on the Caché or GT.M server (*#ewd.time* is an EWD Session value that is created and updated automatically by EWD on every incoming request).

If you need to do anything more sophisticated, such as requesting an EWD fragment and passing it some name/value pairs, you can write your own custom handler, eg:

```
<ext4:container rootPath="/ext-4">

  <script type="text/javascript">
    var setIt = function() {
      var nvp = "action=set&x=123";
      EWD.ajax.getPage({page:'test2cAction',nvp:nvp});
    };
  </script>

  <ext4:panel id="mainPanel" width="600" height="200" title="Toolbar demo">

    <ext4:toolbar dock="top">
      <ext4:button text="Save" handler="function() {Ext.Msg.alert('Test','You clicked the Save but-
ton')}" />
      <ext4:button text="Cancel" />
    </ext4:toolbar>

    <ext4:toolbar dock="bottom" height="50">
      <ext4:layout pack="center" />
      <ext4:button text="Add something" nextPage="test2cPanel" addTo="mainPanel" replacePrevious-
Page="true" />
      <ext4:button text="Set Options" handler="function() {setIt();}"/>
      <ext4:button text="Remove something" />
    </ext4:toolbar>

  </ext4:panel>

</ext4:container>
```

test2cAction.ewd can be nothing more than a fragment with an *onBeforeRender* method, providing an asynchronously-requested fire-and-forget back-end method, eg:

```
<ext4:fragment onBeforeRender="doAction^ext4Demo" />
```

A simple version of the *onBeforeRender* method might look like the following:

```
doAction(sessid)
 n action,x
 s action=$$getRequestValue^%zewdAPI("action",sessid)
 s x=$$getRequestValue^%zewdAPI("x",sessid)
 d trace^%zewdAPI("action="_action_"; x="_x)
 QUIT ""
 ;
```

This is picking up the additional name/value pairs from the incoming request and adding them to EWD's trace global. Of course you could do anything you like at this point within your Caché or GT.M database. The fragment has no content and returns an empty response, so there is no visible effect of this running in your browser's page.

Formatting the Toolbar

ExtJS provides a number of components for formatting your toolbars:

Component	EWD Tag	Purpose
Ext.Toolbar.TextItem	ext4:textItem	adds text into the toolbar
Ext.Toolbar.Fill	ext4:fill	starts right-justification
Ext.Toolbar.Separator	ext4:separator	adds a vertical bar as a separator between toolbar items
Ext.Toolbar.Spacer	ext4:spacer	adds extra horizontal space between toolbar items

For example:

```
  <ext4:toolbar>
    <ext4:textitem text="Some Text" />
    <ext4:separator />
    <ext4:button text="Button 1" />
    <ext4:spacer width="20" />
    <ext4:button text="Button 2" />
    <ext4:fill />
    <ext4:textitem text="More text" />
  </ext4:toolbar>
```

Button Menus

ExtJS buttons don't have to be just simple buttons. They can bring up multi-level menus.

EWD makes it simple to define such menus and make them interact with your Caché or GT.M back-end environment. Menus can be either defined statically, using explicit EWD Custom Tags within your pages or fragments, or dynamically-generated from within Caché or GT.M.

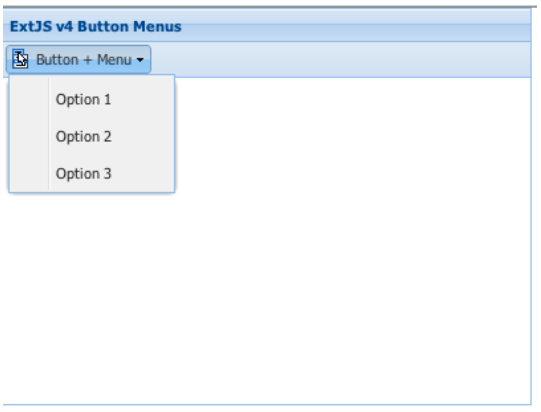
Static Button Menus

A button menu is defined by using the *<ext4:buttonMenu>* tag which is inserted inside an *<ext4:button>* tag. The individual menu items are defined using *<ext4:menuItem>* tags. For example:

```
<ext4:container rootPath="/ext-4" title="Extjs 4 Test">
  <link href="/ext-4/examples/menu/menus.css" />
  <ext4:panel id="mainPanel" title="ExtJS v4 Button Menus" height="300" width="400">
    <ext4:toolbar>
      <ext4:button text="Button + Menu" iconCls="bmenu">
        <ext4:buttonmenu id="mainMenu">
          <ext4:menuitem text="Option 1" />
          <ext4:menuitem text="Option 2" />
          <ext4:menuitem text="Option 3" />
        </ext4:buttonmenu>
      </ext4:button>
    </ext4:toolbar>
  </ext4:panel>
</ext4:container>
```

Note that in this example we've added a special icon to the button. This icon is obtained from the *menus.css* file that is included in the ExtJS distribution. In order to use it, we've added the appropriate *<link>* tag to the Container page. Button icons are optional.

When compiled and run, this page will appear as follows when you click the button:

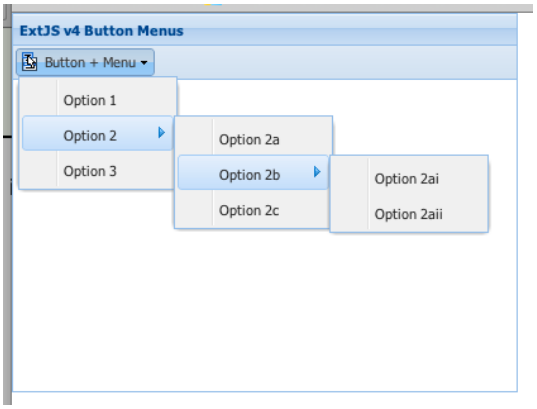


Multi-level Static Menus

You can create as many menu levels as you want. Simply nest <ext4:menuItem> tags inside each other. For example:

```
<ext4:container rootPath="/ext-4" title="Extjs 4 Test">
  <link href="/ext-4/examples/menu/menus.css" />
  <ext4:panel id="mainPanel" title="ExtJS v4 Button Menus" height="300" width="400">
    <ext4:toolbar>
      <ext4:button text="Button + Menu" iconCls="bmenu">
        <ext4:buttonmenu id="mainMenu">
          <ext4:menuItem text="Option 1" />
          <ext4:menuItem text="Option 2">
            <ext4:menuItem text="Option 2a" />
            <ext4:menuItem text="Option 2b">
              <ext4:menuItem text="Option 2ai" />
              <ext4:menuItem text="Option 2aii" />
            </ext4:menuItem>
            <ext4:menuItem text="Option 2c" />
          </ext4:menuItem>
          <ext4:menuItem text="Option 3" />
        </ext4:buttonmenu>
      </ext4:button>
    </ext4:toolbar>
  </ext4:panel>
</ext4:container>
```

In this example above, the menu item Option 2 opens up two further levels:



Adding Interactivity to Menu Items

This is very simple: you use the same techniques that we've previously described for buttons themselves:

- you can use the nextpage attribute in an <ext4:menuItem> tag to fetch a fragment

- you can add a custom 'click' listener to an <ext4:menuItem> tag. Use the listener's first argument to identify the item that has been clicked.

For example:

```
<ext4:menuItem text="Option 1" nextpage="test2cPanel" addTo="mainPanel" replacePreviousPage="true" />

<ext4:menuItem text="Option 3">
  <ext4:listeners>
    <ext4:listener click="function(item,e,eOpts) {Ext.Msg.alert('Test','You selected ' + item.text)}" />
  </ext4:listeners>
</ext4:menuItem>
```

Dynamically-defined Button Menus

EWD adds a very powerful feature: the ability to define a single-level or multi-level button menu from within your Caché or GT.M back-end environment. Your menus can therefore be made context-sensitive, based on information stored in the database or in the user's EWD Session. To create a dynamic button menu, simply add the *sessionName* attribute to the *<ext4:buttonMenu>* tag. The value of the *sessionName* attribute will be the name of an EWD Session Array that you define in the *onBeforeRender* method (normally) for the page or fragment containing the *buttonMenu*.

If you're using the *sessionName* attribute, you don't need to add the *<ext4:buttonMenu>* or *<ext4:menuItem>* tags. All the necessary Javascript will be generated automatically at render-time.

For example:

```
<ext4:container rootPath="/ext-4" title="Extjs 4 Test" onBeforeRender="getButtonMenu^Ext4Demo">
  <link href="/ext-4/examples/menu/menus.css" />
  <ext4:panel id="mainPanel" title="ExtJS v4 Button Menus" height="300" width="400">
    <ext4:toolbar>
      <ext4:button text="Button + Menu" iconCls="bmenu">
        <ext4:buttonmenu id="mainMenu" sessionName="myButtonMenu" />
      </ext4:button>
    </ext4:toolbar>
  </ext4:panel>
</ext4:container>
```

To create a simple menu, the *getButtonMenu() onBeforeRender* method might look like the following:

```
getButtonMenu(sessid)
 n menu
 ;
 s menu(1,"text")="Option 1"
 s menu(2,"text")="Option 2"
 s menu(3,"text")="Option 3"
 d mergeArrayToSession^%zewdAPI(.menu,"myButtonMenu",sessid)
 QUIT ""
 ;
```

So, all that's required is to create an EWD Session Array of the correct name containing the definition of the button menu items. The first subscript of the array is the menu item's position number (starting a 1). The second subscript defines the property name. Naturally enough, the "text" property defines the menu item's visible text. You can define as properties anything that you would have been able to specify as an attribute in the <ext4:menuItem> tags (including any single value Config Option for the *Ext.menu.Item* class. Note that the property names are case-sensitive and must match those of the equivalent Config Option).

Adding Interactivity to Dynamic Button Menus

Clearly our example won't do much other than display the 3 menu options. To add interactivity, simply add the *nextPage*, *addTo* and (if required) the *replacePreviousPage* properties to each item, eg:

```
getButtonMenu(sessid)
 n menu
 ;
 s menu(1,"text")="Option 1"
 s menu(1,"nextPage")="test2cPanel"
 s menu(1,"addTo")="mainPanel"
 s menu(1,"replacePreviousPage")="true"
 s menu(2,"text")="Option 2"
 s menu(3,"text")="Option 3"
 d mergeArrayToSession^%zewdAPI(.menu,"myButtonMenu",sessid)
 QUIT ""
 ;
```

Clearly this can become laborious and unnecessarily repetitive if all or most of the menu options should fetch the same fragment. So you can define the default behaviour within the *<ext4:buttonMenu>* tag, eg:

```
<ext4:container rootPath="/ext-4" title="Extjs 4 Test" onBeforeRender="getButtonMenu^Ext4Demo">
  <link href="/ext-4/examples/menu/menus.css" />
  <ext4:panel id="mainPanel" title="ExtJS v4 Button Menus" height="300" width="400">
    <ext4:toolbar>
      <ext4:button text="Button + Menu" iconCls="bmenu">
        <ext4:buttonmenu id="mainMenu" sessionName="myButtonMenu" nextPage="test3cPanel" addTo="mainPanel"
           replacePreviousPage="true" />
      </ext4:button>
    </ext4:toolbar>
  </ext4:panel>
</ext4:container>
```

These default attributes will now be applied to any menu items in the EWD Session Array that don't specify a nextPage property: ie options 2 and 3 in the example above. Anything defined in the EWD Session Array takes precedence over the defaults.

Dynamically Defining Multi-level Button Menus

To create a multi-level menu, use the special "child" property, eg:

```
getButtonMenu(sessid)
 n menu
 ;
 s menu(1,"text")="Option 1"
 s menu(1,"nextPage")="test2cPanel"
 s menu(1,"addTo")="mainPanel"
 s menu(1,"replacePreviousPage")="true"
 s menu(2,"text")="Option 2"
 s menu(2,"child",1,"text")="Option 2a"
 s menu(2,"child",1,"child",1,"text")="Option 2ai"
 s menu(2,"child",1,"child",2,"text")="Option 2aii"
 s menu(2,"child",2,"text")="Option 2b"
 s menu(3,"text")="Option 3"
 s menu(3,"text")="Option 3"
 d mergeArrayToSession^%zewdAPI(.menu,"myButtonMenu",sessid)
 QUIT ""
 ;
```

Both *Option 2* and *Option 2a* now act as intermediate-level options that open up the lower-level options. By default, *Option 2ai, Option 2aii* and *Option 2b* will fetch the *test3cPanel.ewd* fragment as defined in the *<ext4:buttonMenu>* tag.

You can define menus to any level of nesting you wish.

Identifying the Clicked Menu Item

Clearly we need some way of identifying which menu option was clicked, and in a way that is independent of the nesting of a multi-level menu. EWD allows you to define a set of additional name/value pairs that will be sent with the HTTP request for the fragment when you click a menu option. Just specify an *"nvp"* property for each menu option, eg:

```
getButtonMenu(sessid)
 n menu
 ;
 s menu(1,"text")="Option 1"
 s menu(1,"nextPage")="test2cPanel"
 s menu(1,"addTo")="mainPanel"
 s menu(1,"replacePreviousPage")="true"
 s menu(1,"nvp")="item=1"
 s menu(2,"text")="Option 2"
 s menu(2,"child",1,"text")="Option 2a"
 s menu(2,"child",1,"child",1,"text")="Option 2ai"
 s menu(2,"child",1,"child",1,"nvp")="item=2ai&level=3"
 s menu(2,"child",1,"child",2,"text")="Option 2aii"
 s menu(2,"child",1,"child",2,"nvp")="item=2aii&level=3"
 s menu(2,"child",2,"text")="Option 2b"
 s menu(2,"child",2,"nvp")="item=2b&level=1"
 s menu(3,"text")="Option 3"
 s menu(3,"nvp")="item=3"
 d mergeArrayToSession^%zewdAPI(.menu,"myButtonMenu",sessid)
 QUIT ""
```

You can specify as many name/value pairs as you wish and their names and values are your responsibility. In order to use them, you should use the *$$getRequestValue* EWD API in the *onBeforeRender* method of the fragment fetched by the menu item. You can then define the logic appropriate to the value(s) you received, for example:

```
determineMenuItem(sessid)
 n item,level
 s item=$$getRequestValue^%zewdAPI("item",sessid)
 s level=$$getRequestValue^%zewdAPI("level",sessid)
 i item=2,level=3 d something
 ; etc...
 QUIT ""
 ;
```

Tools

In addition to toolbars, ExtJS allows you to add tools to a panel's header. Tools are small pre-defined icons to which you can add custom handlers, so you can make them behave very similarly to toolbar buttons and they can be a very nice alternative. EWD exposes these tools via the *<ext4:panelTool>* tag.

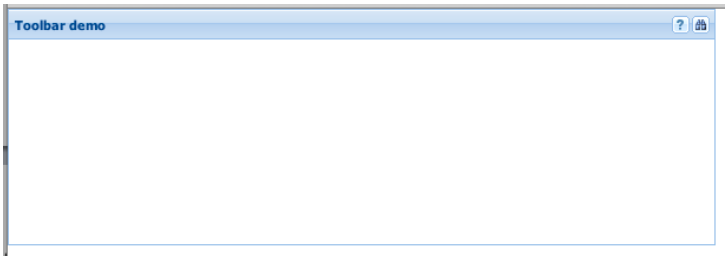
The available tool icons are documented at:

http://docs.sencha.com/ext-js/4-0/#!/api/Ext.panel.Tool-cfg-type

Here's an example:

```
<ext4:container rootPath="/ext-4">

  <ext4:panel id="mainPanel" width="600" height="200" title="Toolbar demo">
    <ext4:paneltool type="help" nextPage="test2cPanel" addTo="mainPanel"
       replacePreviousPage="true" />
    <ext4:paneltool type="search" handler="function() {alert(111)}" />
  </ext4:panel>
</ext4:container>
```

When you compile and run this page, it will look like the following:



Notice the two tool icons in the right corner of the panel's title bar. Clicking them will invoke their respective handlers. As you can see, you use the same techniques as you did with the toolbar buttons to add interactivity.

Note: you can use tools and toolbars together in the same panel if you wish.

Menus

Menu Types

ExtJS provides a number of menu widgets. We've already seen Button Menus, but there are also:

- Independent floating menus (we'll refer to these as just menus) (see the ExtJS class *Ext.menu.Menu*)

- tree menus (see the ExtJS class *Ext.tree.Panel*)

Menus

The menu panels that are attached to Button Menus can be actually be invoked and displayed independently of buttons. In EWD you use the same *<ext4:menuItem>* tags as we saw in the previous chapter, but place them inside an *<ext4:menu>* tag, eg:

```
<ext4:container rootPath="/ext-4">
 <ext4:panel html="Menu test">
  <ext4:menu width="100" height="100" floating="false">
     <ext4:menuitem text="Option 1" />
     <ext4:menuitem text="xOption 2">
      <ext4:listeners>
       <ext4:listener click="alert('option 2 clicked!');" />
      </ext4:listeners>
     </ext4:menuitem>
     <ext4:menuitem text="Option 3">
         <ext4:menuitem text="yOption 3a" />
         <ext4:menuitem text="zOption 3b" />
     </ext4:menuitem>
  </ext4:menu>
 </ext4:panel>
</ext4:container>
```

You can also define the menu dynamically using an EWD Session Array. Once again, the approach is the same as we saw n the previous chapter for the *<ext4:buttonMenu>* tag, eg:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getButtonMenu^ext4Demo">
 <ext4:panel id="panel2" html="Menu test 2">
  <ext4:menu sessionName="myButtonMenu" width="100" height="100" floating="false"
     nextPage="test2cPanel" addTo="panel2" />
 </ext4:panel>
</ext4:container>
```

Tree Menus

Tree menus are very simple to create using EWD and ExtJS. Unlike button menus, they are formally defined in ExtJS using a JSON-based store, so EWD doesn't provide a way of manually defining them using tags to represent the tree menu items. Instead they must always be defined dynamically.

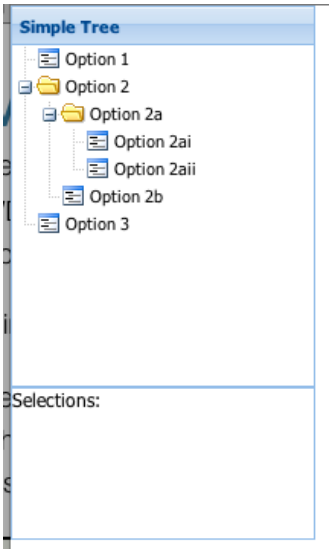
Tree menus are defined using the *<ext4:treePanel>* tag.

In EWD you dynamically define an ExtJS tree menu in exactly the same way as a button menu. EWD will automatically build the tree store from your EWD Session Array that defines the menu. The Session Array has the same syntax, structure and properties as described earlier.

So we could use the *onBeforeRender* method that we used in the previous chapter and render it as a tree menu! Here's an example page that re-uses the button menu Session Array:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getButtonMenu^Ext4Demo">

  <ext4:treepanel title="Simple Tree" id="myTreePanel" height="250" width="200"
    sessionName="myButtonMenu" addTo="mainPanel" replacePreviousPage="true"
    nextpage="test2cPanel" storeId="treeStore" />

  <ext4:panel id="mainPanel" html="Selections:" height="100" width="200" />

</ext4:container>
```

If you compile and run this page, you should see something like the following:



Clicking on the leaf options will fetch the specified fragment and add it to the bottom panel.

You also use the same techniques for identifying which tree menu option was clicked by the user: specify appropriate name/value pairs using the *"nvp"* property for each leaf menu option, and detect them in the onBeforeRender method of the fetched fragment.

Note that you can specify an id for the generated store. This is useful if you want to dynamically modify the tree store contents using the ExtJS APIs at a later stage.

Combining a Tree Panel with a Layout

It's very easy to hook together all the ExtJS components with EWD. For example, we could combine our example tree menu into the layout that we created in an earlier chapter. Here's an example:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getTreeMenu^Ext4Demo">
  <ext4:viewPort layout="border">
    <ext4:panel region="north" id="northPanel" autoheight="true" border="false" margins="0 0 5 0"
      html="<h1 class='x-panel-header'>Layout Demo</h1>" />
    <ext4:treePanel region="west" collapsible="true" title="Navigation" width="150" height="250"
      sessionName="myTreeMenu" border="0" addTo="centerPanel" replacePreviousPage="true"
      nextpage="test4cPanel" />
    <ext4:panel region="south" collapsible="true" title="South Panel" collapsible="true"
      split="true" height="100" minHeight="100" html="This is the South Panel" />
    <ext4:panel region="east" collapsible="true" title="East Panel" split="true" width="150" />
    <ext4:tabPanel region="center" activeTab="0">
      <ext4:panel title="Default Tab" id="centerPanel" />
    </ext4:tabPanel>
  </ext4:viewPort>
</ext4:container>
```

So what we've basically done is to substitute the simple, empty *<ext4:panel>* that we'd defined as the Navigation panel in the west region with our *<ext4:treePanel>*. The *onBeforeRender* method for this example is a minor variation on our buttonMenu example:

```
getTreeMenu(sessid)
 n menu
 ;
 s menu(1,"text")="Option 1"
 s menu(1,"nvp")="item=1"
 s menu(2,"text")="Option 2"
 s menu(2,"child",1,"text")="Option 2a"
 s menu(2,"child",1,"child",1,"text")="Option 2ai"
 s menu(2,"child",1,"child",1,"nvp")="item=2ai&level=3"
 s menu(2,"child",1,"child",2,"text")="Option 2aii"
 s menu(2,"child",1,"child",2,"nvp")="item=2aii&level=3"
 s menu(2,"child",2,"text")="Option 2b"
 s menu(2,"child",2,"nvp")="item=2b&level=2"
 s menu(3,"text")="Option 3"
 s menu(3,"nvp")="item=3"
 d mergeArrayToSession^%zewdAPI(.menu,"myTreeMenu",sessid)
 QUIT ""
```

We've set the default *nextPage* to be the fragment *test4cPanel.ewd* whose content will be added to the layout's panel whose *id* is *centerPanel*:

```
<ext4:treePanel region="west" collapsible="true" title="Navigation" width="150" height="250"
  sessionName="myTreeMenu" border="0" addTo="centerPanel" replacePreviousPage="true"
  nextpage="test4cPanel" />
```

Let's create *test4cPanel.ewd* containing the following:

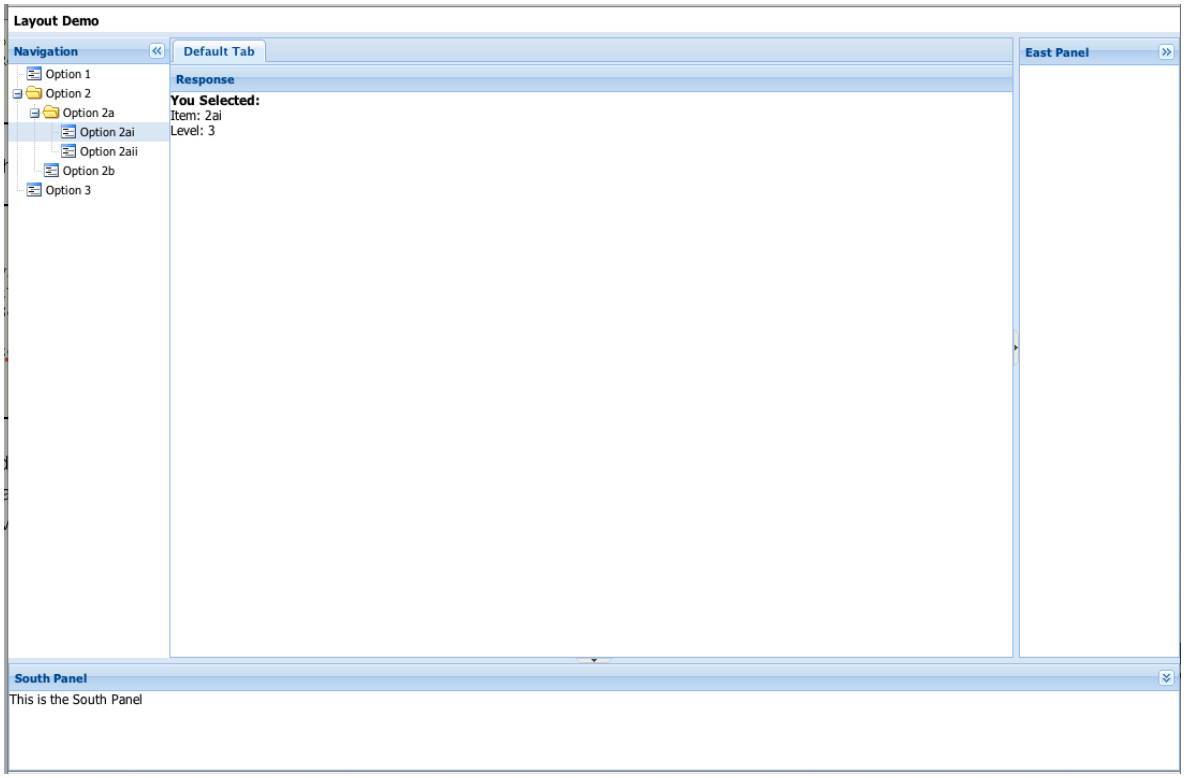
```
<ext4:fragment onBeforeRender="panel4^Ext4Demo">
 <ext4:panel title="Response" border="0" html="#response" />
</ext4:fragment>
```

The *onBeforeRender* method might be as follows:

```
panel4(sessid)
 n html,item,level
 s item=$$getRequestValue^%zewdAPI("item",sessid)
 s level=$$getRequestValue^%zewdAPI("level",sessid)
 s html="&lt;h3>You Selected:&lt;/h3>&lt;div>Item: "_item_"&lt;/div>&lt;div>Level: "_level_"&lt;/
div>"
 d setSessionValue^%zewdAPI("response",html,sessid)
 QUIT ""
 ;
```

This looks rather strange due to a quirk of the way EWD handles fragments. Any **<** characters must be HTML-escaped to prevent them causing Javascript errors. So what this fragment is doing is echoing back the values of the item and level name/value pairs that we've defined in the tree menu items.

If you compile these EWD pages and run the container page, you should see the following layout. In the example below I've expanded the tree and clicked one of the tree menu items which has resulted in the test4cPanel fragment reporting back to me in the center panel.



Now we could have separated out the tree menu into its own fragment. Whether or not you do so is your decision: you may find it easier to manage and maintain you application if it is broken down into small fragments, but on the other hand, each fragment requires the (albeit relatively small) overhead of an HTTP request

Anyway, here's the same application with the tree menu separated out. First the revised Container Page:

```
<ext4:container rootPath="/ext-4">
  <ext4:viewPort layout="border">
    <ext4:panel region="north" id="northPanel" autoheight="true" border="false" margins="0 0 5 0"
html="<h1
class='x-panel-header'>Layout Demo</h1>" />
    <ext4:panel region="west" collapsible="true" title="Navigation" width="150" height="250"
        addPage="layout3Tree" />
    <ext4:panel region="south" collapsible="true" title="South Panel" collapsible="true"
split="true"
height="100" minHeight="100" html="This is the South Panel" />
    <ext4:panel region="east" collapsible="true" title="East Panel" split="true" width="150" />
    <ext4:tabPanel region="center" activeTab="0">
      <ext4:panel title="Default Tab" id="centerPanel" />
    </ext4:tabPanel>
  </ext4:viewPort>
</ext4:container>
```

The West Region panel has been replaced with a simple *<ext4:panel>* container which fetches the *layout3Tree.ewd* fragment. This fragment is as follows. Note that we've moved the *onBeforeRender* method from the Container Page to this fragment:

```
 <ext4:fragment onBeforeRender="getButtonMenu^Ext4Demo">
    <ext4:treePanel width="150" height="250" sessionName="myButtonMenu" border="0"
       addTo="centerPanel" replacePreviousPage="true" nextpage="test4cPanel" />
</ext4:fragment>
```

You should find that this version will run identically to the original.

Tab Panels

Defining a Tab Panel

We've already had a glimpse of a tab panel within the ViewPort example that we've used in earlier chapters. This chapter will look in more detail at how you define and use them within EWD.

An ExtJS Tab Panel is an animated container for other panels. Each panel has a tab associated with it and clicking a panel's tab brings it into view. Only one of the panels inside a tab panel is visible at any one time.

In EWD you define a tab panel using the <ext4:tabPanel> tag. Inside this tag you can define as many <ext4:panel> tags as you wish. Of course those panels can, in turn, contain other ExtJS components, and can automatically fetch fragments containing other ExtJS components.

An Example

Here's our Container Page:

```
<ext4:container rootPath="/ext-4">
  <ext4:viewPort layout="fit">
    <ext4:tabPanel height="200" plain="true" width="450">
      <ext4:panel title="Foo" bodyPadding="10" addPage="test5Panel" />
      <ext4:panel title="Bar" isActive="true" addPage="test5Panel2" >
        <ext4:tabConfig title="Custom Title" tooltip="My custom tooltip!" />
      </ext4:panel>
    </ext4:tabPanel>
  </etx4:viewPort>
</ext4:container>
```

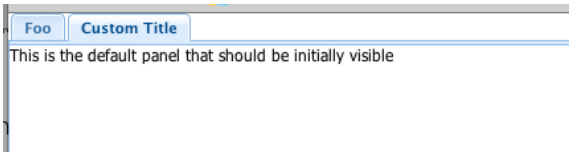
The fragment test5Panel.ewd might contain:

```
<ext4:fragment>
 <ext4:panel border="0" html="Added at <?= #ewd.time ?>" />
</ext4:fragment>
```

The fragment test5Panel2.ewd might contain:

```
<ext4:fragment>
 <ext4:panel border="0" html="This is the default panel that should be initially visible" />
</ext4:fragment>
```

When these pages are compiled and the Container Page is run, it will appear as follows:



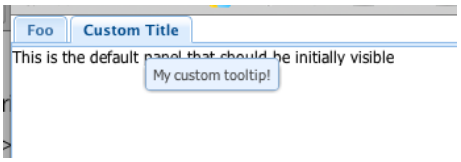
Note the following:

- normally the title attribute of each inner <ext4:panel> tag is used as the text for its tab. This is over-ridden if you add an <ext4:tabConfig> tag

- each inner panel can either contain its own pre-defined content (using child tags that define other ExtJS components), or fetch a fragment containing its content. The example above used the latter technique.

- EWD provides a handy shortcut technique for defining which of the tab panels should display by default (if you don't want to standard default which is for the first tab panel to display). Simply add the attribute *isActive="true"* to the one you want to appear by default. The example above is the equivalent of adding the activeTab="1" attribute to the <ext4:tabPanel> tag. The isActive attribute is a more intuitive description of what you want to achieve.

- You can add tool-tips: small panels that pop up when you move the mouse pointer over a tab. You can use these to provide additional information to the user. The example above demonstrates how the <ext4:tabConfig> tag is used to define a tool tip. Here's what the tool-tip looks like when it pops up:



Adding Dynamic Behaviour

ExtJS allows you do many dynamic things with tabs and tab panels, including:

- adding panels to and removing panels from a tab panel

- programmatically selecting tabs

Here's an example of how you can perform these actions via EWD:

```
<ext4:container rootPath="/ext-4">

   <ext4:tabPanel height="200" width="450" id="myTabPanel">
      <ext4:panel title="Foo" bodyPadding="10" html="Foo Panel" isActive="true" id="tab0" />
      <ext4:panel title="Bar" html="Bar Panel" id="tab1" />
    </ext4:tabPanel>

    <ext4:button text="Select 2nd" handler="function() {Ext.getCmp('myTabPanel').setActiveTab(1)}" />
    <ext4:button text="Add Tab" nextPage="newTab" addTo="myTabPanel" />
    <ext4:button text="Remove 1st" handler="function() {Ext.getCmp('myTabPanel').remove(0)}" />

</ext4:container>
```

Here's the new tab fragment:

```
<ext4:fragment onBeforeRender="getTabName^Ext4Demo">
 <ext4:panel title="#tabName" id="#tabName" html="New tab <?= #tabNo ?> added at <?= #ewd.time ?>" />
</ext4:fragment>
```

Note that is is critically important to keep the id values of all ExtJS components unique: if you don't, the effects can be unpredictable. Hence, this fragment generates a unique id for itself within its onBeforeRender method as follows:

```
getTabName(sessid)
 ;
 n tabNo
 ;
 s tabNo=$$getSessionValue^%zewdAPI("tabNo",sessid)
 d setSessionValue^%zewdAPI("tabNo",tabNo+1,sessid)
 d setSessionValue^%zewdAPI("tabName","NewTab"_(tabNo+1),sessid)
 QUIT ""
 ;
```

Try compiling and running this example to see it in action.

Windows

ExtJS Windows

The ExtJS documentation describes their Window components as follows:

"A specialized panel intended for use as an application window. Windows are floated, resizable, and draggable by default. Windows can be maximized to fill the viewport, restored to their prior size, and can be minimized."

As such, they are an important part of the ExtJS UI feature set.

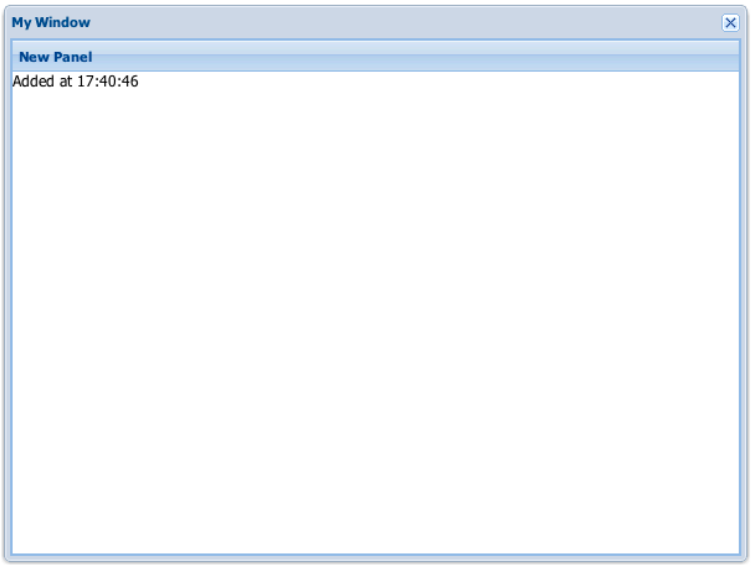
Once again, EWD makes them easy to use and integrate with Caché and GT.M.

Simple Example

Here's a simple Container Page that creates a Window and populates it with a panel that is delivered from one of the fragments we've previously developed.

```
<ext4:container rootPath="/ext-4">
  <ext4:window title="My Window" height="450" width="600" layout="fit" autoShow="true" addPage="test2cPanel" />
</ext4:container>
```

When compiled and run, this will appear as follows:



You'll find that you can drag the window around, and clicking on the X in the top right will make it disappear.

If you add an id attribute to the <ext4:window> tag, you'll then be able to invoke any of the methods available for windows, as described in the ExtJS documentation.

-

Modal Windows

You can create a modal window very easily too. When a modal window appears, everything behind it becomes greyed out and inaccessible. Only when the modal window is closed does the user gain access to the rest of the browser content. Here's an example:

```
<ext4:container rootPath="/ext-4">
  <ext4:viewPort object="theViewPort" layout="border" var="true">
    <ext4:modalwindow title="My Window" height="200" width="400" layout="fit" autoShow="true"
        addPage="test2cPanel" />
  </ext4:viewPort>
</ext4:container>
```

Grids

ExtJS Grids

Grids are probably the most powerful feature within ExtJS, allowing amazing flexibility in terms of both display and editing of data. As with all the other ExtJS widgets and components, EWD makes grids straightforward to work with and almost trivial to integrate with Caché and GT.M.

You create an ExtJS grid by using the *<ext4:gridpanel>* tag. Grids use data stored in a JSON store, but, as with Tree Menu stores, EWD allows you to define your grid data in a simple EWD Session Array: EWD converts this to the necessary JSON structure and defines and creates the data store at page/fragment render-time.

You also have to define the columns that will be used for the grid. You have two ways of doing this:

- explicitly, using *<ext4:gridcolumn>* tags

- programmatically by defining an EWD Session Array

Simple Example

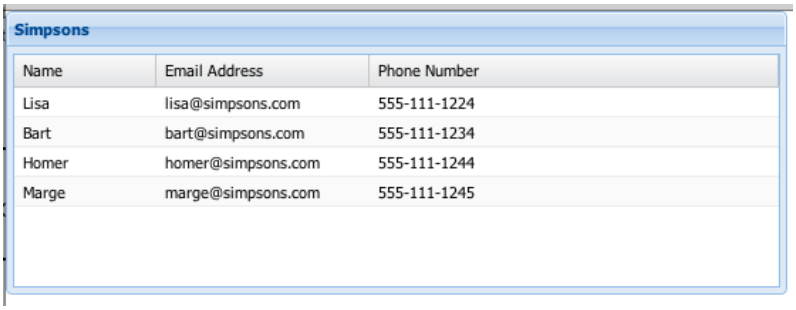
The following example creates a simple display-only grid, using text values. The columns are defined explicitly using tags, and the data is a simple EWD Session Array. Here's the Container Page:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons">
    <ext4:gridcolumn text="Name" width="100" sortable="false" hideable="false" dataIndex="name" />
    <ext4:gridcolumn text="Email Address" width="150" dataIndex="email" />
    <ext4:gridcolumn text="Phone Number" flex="1" dataIndex="phone" />
  </ext4:gridPanel>

</ext4:container>
```

The *onBeforeRender* method is as follows:

```
getGridData(sessid)
 ;
 n data
 ;
 s data(1,"name")="Lisa"
 s data(1,"email")="lisa@simpsons.com"
 s data(1,"phone")="555-111-1224"
 s data(2,"name")="Bart"
 s data(2,"email")="bart@simpsons.com"
 s data(2,"phone")="555-111-1234"
 s data(3,"name")="Homer"
 s data(3,"email")="homer@simpsons.com"
 s data(3,"phone")="555-111-1244"
 s data(4,"name")="Marge"
 s data(4,"email")="marge@simpsons.com"
 s data(4,"phone")="555-111-1245"
 d deleteFromSession^%zewdAPI("simpsons",sessid)
 d mergeArrayToSession^%zewdAPI(.data,"simpsons",sessid)
 ;
 QUIT ""
```

if you compile and run this Container Page, it should look like the following:



The structure of the EWD Session Array is as follows:

data(rowNumber, dataIndex) = cellValue

You'll notice how the *dataIndex* attribute of the *<ext4:gridColumn>* tags maps to the corresponding dataIndex name within the EWD Session Array.

You can control the behaviour and characteristics of each column by using the grid column attributes. These map directly to the Config Options of the *Ext.grid.column.Column* class, so you should consult the ExtJS documentation and examples for more information.

Dynamic Columns

Here's the same example as above, but now the columns are also defined within the onBeforeRender method. You'll see that the column array properties map directly to the <ext4:gridColumn> tags and the Ext.grid.column.Column Config Options:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridDef^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
columnDefinition="colDef" />

</ext4:container>
```

The *onBeforeRender* method is as follows:

```
getGridDef(sessid)
 ;
 n column,data
 ;
 s column(1,"dataIndex")="name"
 s column(1,"text")="Name"
 s column(1,"width")=100
 s column(1,"sortable")="false"
 s column(1,"hideable")="false"
 ;
 s column(2,"dataIndex")="email"
 s column(2,"text")="Email Address"
 s column(2,"width")=150
 ;
 s column(3,"dataIndex")="phone"
 s column(3,"text")="Phone Number"
 s column(3,"flex")=1
 ;
 d deleteFromSession^%zewdAPI("colDef",sessid)
 d mergeArrayToSession^%zewdAPI(.column,"colDef",sessid)
 ;
 s data(1,"name")="Lisa"
 s data(1,"email")="lisa@simpsons.com"
 s data(1,"phone")="555-111-1224"
 s data(2,"name")="Bart"
 s data(2,"email")="bart@simpsons.com"
 s data(2,"phone")="555-111-1234"
 s data(3,"name")="Homer"
 s data(3,"email")="homer@simpsons.com"
 s data(3,"phone")="555-111-1244"
 s data(4,"name")="Marge"
 s data(4,"email")="marge@simpsons.com"
 s data(4,"phone")="555-111-1245"
 d deleteFromSession^%zewdAPI("simpsons",sessid)
 d mergeArrayToSession^%zewdAPI(.data,"simpsons",sessid)
 ;
 QUIT ""
 ;
```

Use the columnDefinition attribute in the <ext4:gridPanel> tag to specify the EWD Session Array that is to be used to define the Grid Columns. The Column Definition Session Array is structured as follows:

column(columnNumber, configOptionName) = Grid Column Config Option Value

Each *dataIndex* subscript in the data array should have a corresponding column array node with a matching *dataIndex* value, eg the following means that the email values will be placed into the second column:

```
s column(2,"dataIndex")="email"
s data(1,"email")="lisa@simpsons.com"
s data(2,"email")="bart@simpsons.com"
s data(3,"email")="homer@simpsons.com"
s data(4,"email")="marge@simpsons.com"
```

Note that both the column numbers and row numbers start from 1.

Identifying the Grid and Store

It's always a good idea to provide your own id for your grids, and also to provide an id for the store. By doing so you can dynamically manipulate and modify the grid using its built-in methods, and also dynamically modify the store contents if required. We'll extend our example grid as follows:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridDef^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
columnDefinition="colDef" id="myGrid" storeId="myStore" />

</ext4:container>
```

We can now set a pointer to our GridPanel instance and the data store using, respectively:

Ext.getCmp('myGrid');

Ext.getCmp('myStore');

Special Column Types

EWD provides mappings for both explicitly-defined columns and dynamically-defined columns for the special ExtJS Grid column types as follows:

Column Type	ExtJS Class	EWD Tag	xtype	Purpose
Boolean	Ext.grid.column.Boolean	ext4:booleanColumn	booleancolumn	Renders boolean data fields
Date	Ext.grid.column.Date	ext4:dateColumn	datecolumn	Renders date fields according to a format string
Number	Ext.grid.column.Number	ext4:numberColumn	numbercolumn	Renders number data fields
Action	Ext.grid.column.Action	ext4:actionColumn	actioncolumn	Renders one or more icons in a grid cell
Row Numberer	Ext.grid.RowNumberer	ext4:rowNumberer	rownumberer	Creates a special column that provides automatic row numbering

Explicity-defined Example

The previous simple grid has been extended to include an example of each of the special column types as follows:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData2^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="700" sessionName="simpsons">
    <ext4:rowNumberer />
    <ext4:gridcolumn text="Name" width="50" sortable="false" hideable="false" dataIndex="name" />
    <ext4:gridcolumn text="Email Address" width="150" dataIndex="email" />
    <ext4:gridcolumn text="Phone Number" width="120" dataIndex="phone" />
    <ext4:booleancolumn text="Interesting" dataIndex="interest" trueText="Yes" falseText="No" />
    <ext4:datecolumn text="DOB" dataIndex="dob" format="F d Y" width="120" />
    <ext4:numbercolumn text="Height" dataIndex="height" format="|0.00" width="50" />
    <ext4:actioncolumn width="50">
      <ext4:icon icon="/ext-4/examples/shared/icons/fam/cog_edit.png" tooltip="Edit" nextpage="test2cPanel"
         addTo="mainPanel" replacePreviousPage="true" nvp="iconNo=1" />
      <ext4:icon icon="/ext-4/examples/shared/icons/fam/delete.gif" tooltip="Delete"
        handler="function(grid, rowIndex, colIndex) {alert('delete: record=' +
        EWD.ext4.getGridRowNo(grid,rowIndex));}" />
    </ext4:actioncolumn>
  </ext4:gridPanel>

  <ext4:panel id="mainPanel" height="200" width="200" />

</ext4:container>
```

The onBeforeEdit method for the example above is as follows:

```
getGridData2(sessid)
 ;
 n data
 ;
 s data(1,"name")="Lisa"
 s data(1,"email")="lisa@simpsons.com"
 s data(1,"phone")="555-111-1224"
 s data(1,"interest")="true"
 s data(1,"dob")="06/20/1995"
 s data(1,"height")="1.2134"
 s data(2,"name")="Bart"
 s data(2,"email")="bart@simpsons.com"
 s data(2,"phone")="555-111-1234"
 s data(2,"interest")="true"
 s data(2,"dob")="02/04/1991"
 s data(2,"height")="1.5"
 s data(3,"name")="Homer"
 s data(3,"email")="homer@simpsons.com"
 s data(3,"phone")="555-111-1244"
 s data(3,"interest")="false"
 s data(3,"dob")="09/10/1956"
 s data(3,"height")="1.92"
 s data(4,"name")="Marge"
 s data(4,"email")="marge@simpsons.com"
 s data(4,"phone")="555-111-1245"
 s data(4,"interest")="false"
 s data(4,"dob")="11/20/1960"
 s data(4,"height")="1.75"
 d deleteFromSession^%zewdAPI("simpsons",sessid)
 d mergeArrayToSession^%zewdAPI(.data,"simpsons",sessid)
 ;
 QUIT ""
 ;
```

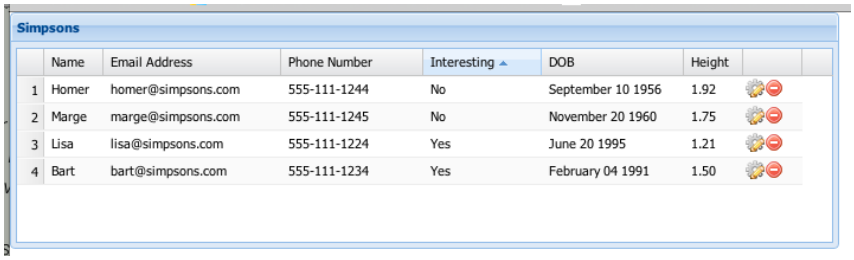
Note: if you look carefully at the <ext4:numberColumn> tag in the example above, you'll see that the format attribute value is:

format="|0.00"

That leading | character serves an important purpose: it overrides EWD's normal attribute quoting rules that would see the value as being numeric and hence not requiring quotes. The format attribute defines a pattern that has to be treated as a string value.
EWD has two special overrides for attribute values:

- *if the first character is | (ie vertical bar), the value will be treated as a quoted string*
- *if the first character is . (ie a period or full-stop), the value will be unquoted*

If you compile and run the Container Page above, you should see:



Some key features of this example:

- The Boolean Column type allows you to transform boolean data (eg true/false) into an alternative format for display (eg Yes/No)

- The available options for the formats for Date Columns are provided in the ExtJS documentation. See the *Ext.Date* class

- Your Action Columns should contain one or more icons to which you'll assign handler functions. Since you'll probably want to perform your processing at the back-end, it's critically important that you can identify the row to which the clicked icon belongs. To complicate matters, the user may have clicked on one or more column headings and re-ordered the rows, so you need to identify the original row number so that you can correlate it back to your Grid Data Session Array. EWD makes this simple: it automatically adds a hidden column to your Grids that contains the original row number, and also provides you with a function to get the value of this field from within your Action Column icon handlers:

    - *EWD.ext4.getGridRowNo(grid, rowIndex)*

  You can send this back as an additional name/value pair along with a request for a fragment, eg:

```
handler="function(grid, rowIndex) {var nvp = 'rowNo=' + EWD.ext4.getGridRowNo(grid, rowIndex);
EWD.ajax.getPage({page:'editRowData',nvp:nvp});}"
```

- However, in most situations you can make use of the *nextPage* attribute, as used by the first icon in our example:

```
<ext4:icon icon="/ext-4/examples/shared/icons/fam/cog_edit.png" tooltip="Edit2"
  nextpage="test2cPanel" addTo="mainPanel" replacePreviousPage="true" nvp="iconNo=1" />
```

- This provides you with an even simpler mechanism that can be used to fetch a fragment that can add itself to a panel, as in the example above. Use the optional *nvp* attribute if you need to uniquely identify the request as coming from a particular icon (eg if all the icons have the same nextpage value). If you want the fragment to run silently or if the fragment has nothing but an *onBeforeRender* method, don't specify the *addTo* or *replacePreviousPage* attributes. The *nextpage* attribute therefore works identically to how we saw it work for buttons earlier. EWD recognises that you're using *nextPage* with an ActionColumn's *icon*, and adds the row number to the name/value pairs that are sent with the request to the back end. This name/value pair is named *row*, and you can pick it up within the fragment's *onBeforeRender* method using:

    - s row=$$getRequestValue^%zewdAPI("row",sessid)

- Note that EWD ensures that the row number is the original row number, irrespective of whether or not the user has re-ordered the Grid rows by clicking on column headers.

Dynamically-defined Example

Here's our Grid example defined dynamically:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridDef2^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" id="myGrid" storeId="myStore" frame="true" height="200" width="700"
sessionName="simpsons" columnDefinition="colDef" />

</ext4:container>
```

You'll see that very little has changed as far as the Container Page is concerned, except that we've now dispensed with the *<ext4:gridColumn>* tags. All the changes are in the onBeforeRender method which now includes the definition of the special column types. Here's what it contains:

```
getGridDef2(sessid)
 ;
 n column,data
 ;
 s column(1,"xtype")="rownumberer"
 ;
 s column(2,"dataIndex")="name"
 s column(2,"text")="Name"
 s column(2,"width")=50
 s column(2,"sortable")="false"
 s column(2,"hideable")="false"
 ;
 s column(3,"dataIndex")="email"
 s column(3,"text")="Email Address"
 s column(3,"width")=150
 ;
 s column(4,"dataIndex")="phone"
 s column(4,"text")="Phone Number"
 s column(4,"width")=120
 ;
 s column(5,"dataIndex")="interest"
 s column(5,"text")="Interesting"
 s column(5,"xtype")="booleancolumn"
 s column(5,"trueText")="Yes"
 s column(5,"falseText")="No"
 ;
 s column(6,"dataIndex")="dob"
 s column(6,"text")="DOB"
 s column(6,"xtype")="datecolumn"
 s column(6,"format")="F d Y"
 s column(6,"width")="120"
 ;
 s column(7,"dataIndex")="height"
 s column(7,"text")="Height"
 s column(7,"xtype")="numbercolumn"
 s column(7,"format")="|0.000"
 s column(7,"width")="50"
 ;
 s column(8,"xtype")="actioncolumn"
 s column(8,"width")="50"
 s column(8,"icon",1,"icon")="/ext-4/examples/shared/icons/fam/cog_edit.png"
 s column(8,"icon",1,"tooltip")="Edit"
 s column(8,"icon",1,"nextPage")="test2cPanel"
 s column(8,"icon",1,"addTo")="mainPanel"
 s column(8,"icon",1,"replacePreviousPage")="true"
 s column(8,"icon",2,"icon")="/ext-4/examples/shared/icons/fam/delete.gif"
 s column(8,"icon",2,"tooltip")="Delete"
 s column(8,"icon",2,"handler")="function(grid, rowIndex, colIndex) {alert('delete: record=' +
EWD.ext4.getGridRowNo(grid,rowIndex));}"
 ;
 d deleteFromSession^%zewdAPI("colDef",sessid)
 d mergeArrayToSession^%zewdAPI(.column,"colDef",sessid)
 ;
 s data(1,"name")="Lisa"
 s data(1,"email")="lisa@simpsons.com"
 s data(1,"phone")="555-111-1224"
 s data(1,"interest")="true"
 s data(1,"dob")="06/20/1995"
 s data(1,"height")="1.2134"
 s data(2,"name")="Bart"
 s data(2,"email")="bart@simpsons.com"
 s data(2,"phone")="555-111-1234"
 s data(2,"interest")="true"
 s data(2,"dob")="02/04/1991"
 s data(2,"height")="1.5"
 s data(3,"name")="Homer"
 s data(3,"email")="homer@simpsons.com"
 s data(3,"phone")="555-111-1244"
 s data(3,"interest")="false"
 s data(3,"dob")="09/10/1956"
 s data(3,"height")="1.92"
 s data(4,"name")="Marge"
 s data(4,"email")="marge@simpsons.com"
 s data(4,"phone")="555-111-1245"
 s data(4,"interest")="false"
 s data(4,"dob")="11/20/1960"
 s data(4,"height")="1.75"
 d deleteFromSession^%zewdAPI("simpsons",sessid)
 d mergeArrayToSession^%zewdAPI(.data,"simpsons",sessid)
 ;
 QUIT ""
```

So you can see that you specify the special column types by using their *xtypes* (as listed in the table at the start of this section). Otherwise you should see that there is a one-to-one correspondence between the explicit tag attributes and the

EWD Session Array subscripts. Note, however, that whilst the tag attribute names are case-insensitive, the Session Array attribute names are case-sensitive and must match the documented ExtJS class Config Options.

You can probably see that completely dynamically-defined Grids are very powerful: if wrapped as a fragment, you have the basis of a re-usable grid component whose columns and content is completely variable, dependent on when and how it is fetched.

On the other hand, explicitly-defined Grids are arguably easier to understand and maintain: their structure and content isn't defined in programmatic logic in the back-end.

Which you use is up to you, and will depend on a balance of factors that you must decide upon.

Editable Grids

ExtJS grids aren't simply read-only widgets: you can allow cells to be edited, and you can add your own custom validation logic. You'll find examples in the ExtJS documentation of editable grids, and you can use the direct EWD/ExtJS tag mappings to create the equivalent techniques in your EWD pages.

However, EWD provides a simple set of shortcut mechanisms that makes it much simpler to define editable grids and to define validation that is to be conducted asynchronously within the GT.M or Caché back-end. Behind the scenes, EWD will create the same Javascript code that you'll see in those ExtJS examples.

The Editor Tag

The low-level way of making cells editable in a grid is to define an editor for a column. Once this is done, all cells in that column will be editable. ExtJS provides a number of editor types, including:

- textfield

- numberfield

- datefield

- combobox

As you'll see, EWD provides a shortcut technique for enabling each of these editor types. The shortcut techniques use a set of typical default settings which will be satisfactory in most cases. However, if you need to use different settings, you should revert to the low-level approach for editing. To do this, you use the *<ext4:editor>* tag inside the appropriate *<ext4:gridColumn>* tag. The *<ext4:editor>* tag maps directly to the editor Config Option of the *Ext.grid.column.Column* class. You also need to use a *celllediting* plugin for the Grid Panel. If this sounds complex and confusing, it is! However, if you're working from the ExtJS examples, you'll find that you can use the standard technique of mapping from the ExtJS components and their Config Options to correspondingly-named EWD tags and attributes. This low-level use is beyond the scope of this document.

Fortunately, for most situations, the built-in EWD shortcut mechanisms for defining editable grids will be adequate for your needs: let EWD do all that hard work for you, leaving you to specify your editing requirements in an intuitive and simple way. These shortcut techniques are described below:

Editing using the TextField Editor

The simplest form of grid cell editing is to treat the cell as a text field and use the built-in ExtJS *textfield* editor. Here's our earlier simple Grid example using explicitly-defined columns, but modified to allow the Name cells to be editable:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
       validationPage="gridValidateTest">
    <ext4:gridcolumn text="Name" width="100" sortable="false" editAs="textfield" hideable="false"
dataIndex="name"/>
    <ext4:gridcolumn text="Email Address" width="150" dataIndex="email" />
    <ext4:gridcolumn text="Phone Number" flex="1" dataIndex="phone" />
  </ext4:gridPanel>

</ext4:container>
```

So you use the *editAs* attribute within any column that you want to be editable. The value of the *editAs* attribute defines the type of ExtJS editor to use for the field. By default, the user needs to click twice on the field to trigger the editor. However you can override this by adding the attribute *clicksToEdit* to the *<ext4:gridPanel>* tag, eg to trigger the editor simply by tapping on a Name cell:

```
<ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
      validationPage="gridValidateTest" clicksToEdit="1" >
```

In order to validate the edited value at the back-end, you specify a fragment name using the *validationPage* attribute: that fragment's responsibility will be to provide an *onBeforeRender* method that performs the necessary tests.

Here's all there is to our validation fragment (in our example it will be named *gridValidateTest.ewd*):

```
<ext4:fragment onBeforeRender="gridValidate^Ext4Demo" />
```

Its *onBeforeRender* method is as follows:

```
gridValidate(sessid)
 ;
 n colName,rowNo,value
 ;
 s rowNo=$$getRequestValue^%zewdAPI("row",sessid)
 s colName=$$getRequestValue^%zewdAPI("colName",sessid)
 s value=$$getRequestValue^%zewdAPI("value",sessid)
 ; ... perform the necessary checks, eg
 i value=1234 QUIT $$gridValidationFail^%zewdExt4Code(sessid,"Thats an invalid value!","Rubbish")
 QUIT $$gridValidationPass^%zewdExt4Code()
 ;
```

The key features of a validation method are:

- the edited values are sent as request values, so you access them using the *$$getRequestValue* API
- the available request values are:
    - ***row***: the original row number (ie it doesn't matter if the user has re-ordered the grid by clicking a column heading)
    - ***colName***: the column name property
    - ***value***: the edited value of the cell
    - ***originalValue***: the value of the cell prior to the edit

The *row* and *colName* provide you with the pointers you need for the corresponding Grid Data Session Array, in this case a Session Array named *simpsons*.

How you validate the edited field is entirely up to you, but as far as EWD is concerned there are only two possible outcomes:

- validation passed

- validation failed

If validation passed, you'll typically want to just silently accept the edit. To do this, you Quit from the onBeforeRender method, returning the value of a special function:

$$gridValidationPass^%zewdExt4Code()

If validation failed, you'll typically want to alert the user, giving them some information on what they did wrong. To do this, you Quit from the onBeforeRender method, returning the value of another special function:

$$gridValidationFail^%zewdExt4Code(sessid, reasonText, titleText)

The *reasonText* is the optional alert message text that instructs the user what the error was. If you don't define *reasonText*, the message will default to *"Invalid value: xxxx"* where *xxx* is the edited value.

The *titleText* allows you to optionally provide a title for the alert panel. If you don't define titleText, it defults to *"Validation Error"*

You now have a simple editable grid. Try out the example, and try modifying the validation onBeforeRender method to familiarise yourself with how it works and how you can customise your validation according to own requirements.

Using Dynamically-defined Columns

EWD provides an analogous method for Grids where you've defined your columns programmatically. Simply add the *editas* property to the column definition array for the required column, eg we could re-define the previous example as follows:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridDef^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
columnDefinition="colDef" clicksToEdit="1" validationPage="gridValidateTest" />

</ext4:container>
```

The column definition part of this page's onBeforeRender method would look like this:

```
 s column(1,"dataIndex")="name"
 s column(1,"text")="Name"
 s column(1,"width")=100
 s column(1,"sortable")="false"
 s column(1,"hideable")="false"
 s column(1,"editas")="textfield"
 ;
 s column(2,"dataIndex")="email"
 s column(2,"text")="Email Address"
 s column(2,"width")=150
 ;
 s column(3,"dataIndex")="phone"
 s column(3,"text")="Phone Number"
 s column(3,"flex")=1
 ;
 d deleteFromSession^%zewdAPI("colDef",sessid)
 d mergeArrayToSession^%zewdAPI(.column,"colDef",sessid)
```

That's all that's required: this will now behave identically to the explicitly-defined example.

Numberfield Editor

ExtJS also includes a special number editor capability for grid cells that contain numbers, using a spinner control. Typically you'll use this in conjunction with columns that are defined as *numberColumns*.

To use the *numberfield* editor with explicitly-defined columns, you'd simply add the following column definition to your grid:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
       validationPage="gridValidateTest">
    ..... etc

    <ext4:numbercolumn text="Height" width="50" editAs="numberfield" format="|0.00" dataIndex="height" />

  </ext4:gridPanel>

</ext4:container>
```

If you're using dynamically-defined columns, here's how you'd specify the same thing:

```
s column(7,"dataIndex")="height"
s column(7,"text")="Height"
s column(7,"xtype")="numbercolumn"
s column(7,"format")="|0.000"
s column(7,"width")="50"
s column(7,"editas")="numberfield"
```

You may find that you need to take more fine-grained control of the *numberfield* editor. If so, you can drop down a level and use the *<ext4:editor>* tag. For example:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
       validationPage="gridValidateTest">
    ..... etc

    <ext4:numbercolumn text="Height" dataIndex="height" format="|0.00" width="50">
      <ext4:editor xtype="numberfield" allowBlank="false" minValue="1" maxValue="3" step="0.1" />
    </ext4:numbercolumn>

  </ext4:gridPanel>
```

To do the same thing using dynamic columns:

```
s column(7,"dataIndex")="height"
s column(7,"text")="Height"
s column(7,"xtype")="numbercolumn"
s column(7,"format")="|0.000"
s column(7,"width")="50"
s column(7,"editas")="numberfield"
s column(7,"editor","allowBlank")="false"
s column(7,"editor","minValue")="1"
s column(7,"editor","maxValue")="3"
s column(7,"editor","step")=0.01
```

Back-end validation works identically to the textfield editor, as described earlier.

Datefield Editor

ExtJS also includes a very cool date editor capability for grid cells that contain dates, complete with a calendar pop-up. Typically you'll use this in conjunction with columns that are defined as *dateColumns*.

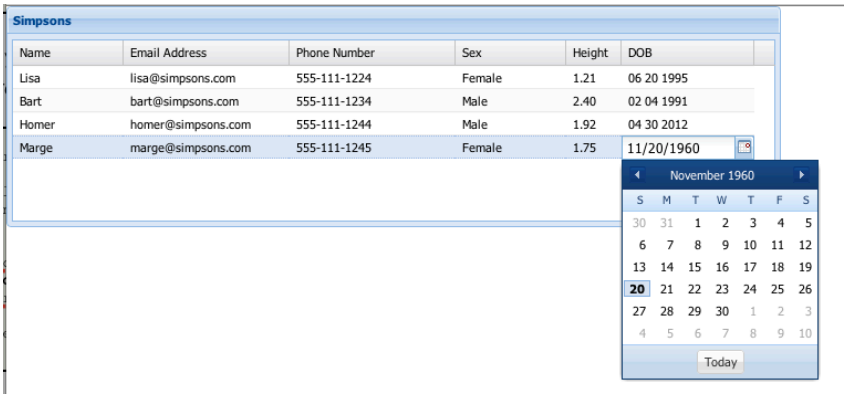
To use the *datefield* editor with explicitly-defined columns, you'd simply add the following column definition to your grid:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
       validationPage="gridValidateTest">
    ..... etc

    <ext4:datecolumn text="DOB" dataIndex="dob" format="m d Y" width="120" editas="datefield" />

  </ext4:gridPanel>

</ext4:container>
```

If you're using dynamically-defined columns, here's how you'd specify the same thing:

```
s column(5,"dataIndex")="dob"
s column(5,"text")="DOB"
s column(5,"width")=120
s column(5,"format")= "m d Y"
s column(5,"editas")="datefield"
s column(5,"xtype")="datecolumn"
```

Here's an example of the datefield editor in action within a grid:



If you need more fine-grained control over the editor, or need to use Config Option values for the editor other than the defaults used by EWD, you can drop down a level and use the *<ext4:editor>* tag, in a way similar to the example for the *numberfield* described above.

Back-end validation works identically to the *textfield* editor, as described earlier. Note that the *value* and *originalValue* format for dates sent by EWD to the back-end is *mm/dd/yyyy* , eg:

```
value: 04/30/2012
```

Combobox Editor

ExtJS also includes a combo box editor capability for grid cells that contain one of a pre-determined list of possible values.

To use the *combobox* editor with explicitly-defined columns, you'd simply add the following column definition to your grid:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
       validationPage="gridValidateTest">
    ..... etc

    <ext4:gridcolumn text="Sex" width="100" dataIndex="sex" editas="combobox">
       <ext4:options>
         <ext4:option value="m" displayValue="Male" />
         <ext4:option value="f" displayValue="Female" />
       </ext4:options>
    </ext4:gridcolumn>

  </ext4:gridPanel>

</ext4:container>
```

You can alternatively make use of a standard EWD List by using the *useList* attribute:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
       validationPage="gridValidateTest">
    ..... etc

    <ext4:gridcolumn text="Sex" width="100" dataIndex="sex" editas="combobox" useList="sex" />

  </ext4:gridPanel>

</ext4:container>
```

In this case, the list would be defined within the page/fragment's *onBeforeRender* method in the standard way, eg:

```
d clearList^%zewdAPI("sex",sessid)
d appendToList^%zewdAPI("sex","Male","m",sessid)
d appendToList^%zewdAPI("sex","Female","f",sessid)
```

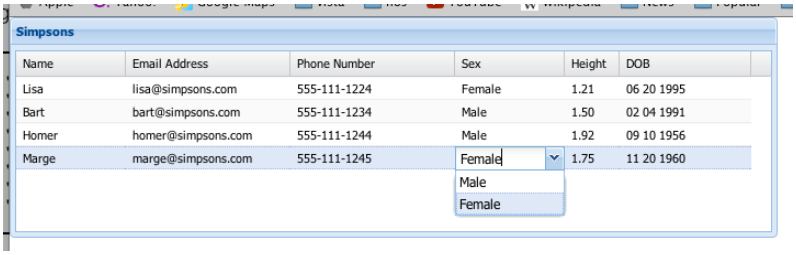
If you're using dynamically-defined columns, here's how you'd specify the same thing:

```
s column(4,"dataIndex")="sex"
s column(4,"text")="Sex"
s column(4,"width")=100
s column(4,"editas")="combobox"
s column(4,"options",1,"value")="m"
s column(4,"options",1,"displayValue")="Male"
s column(4,"options",2,"value")="f"
s column(4,"options",2,"displayValue")="Female"
```

Or, to use an EWD list:

```
s column(4,"dataIndex")="sex"
s column(4,"text")="Sex"
s column(4,"width")=100
s column(4,"editas")="combobox"
s column(4,"useList")="sex"
```

Here's an example of what the Combobox editor looks like with an ExtJS Grid:



Back-end validation works identically to the textfield editor, as described earlier. EWD sends through to the back-end the *value* property of the drop-down list, rather than the *displayValue.*

Grouping

ExtJS allows you to group all the values of a particular column. EWD makes this simple to control. For explicitly-defined columns, just add the attribute *groupField="true"* to the column tag that you want to group by, eg:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridData^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" frame="true" height="200" width="550" sessionName="simpsons"
       validationPage="gridValidateTest">
    ..... etc

    <ext4:gridcolumn text="Email Address" width="150" dataIndex="email" groupField="true" />

  </ext4:gridPanel>

</ext4:container>
```

If you're using dynamically-defined columns, you first add the *groupField* property to the appropriate column definition, eg:

```
 s column(2,"dataIndex")="email"
 s column(2,"text")="Email Address"
 s column(2,"width")=150
 s column(2,"groupField")="true"
```

Second, you must also specify in the *<ext4:gridPanel>* tag that you want grouping enabled, eg:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getGridDef3^Ext4Demo" >

  <ext4:gridPanel title="Simpsons" id="simpsonsGrid" frame="true" height="200" width="700" sessionName="simpsons"
clicksToEdit="1" validationPage="gridValidateTest" columnDefinition="colDef" grouping="true" />

</ext4:container>
```

Note: this *grouping* attribute is unnecessary when you're using explicitly defined column tags, because EWD's compiler is able to add it automatically to the *<ext4:gridPanel>* tag for you.

Forms

ExtJS Forms

ExtJS provides a set of very nicely styled variants of the standard HTML form fields, in addition to some additional enhanced form widgets. EWD makes them easy to use and automatically integrates them with your back-end database.

The following is a simple example, with just one single text field:

```
<ext4:container rootPath="/ext-4" onBeforeRender="getFormData^Ext4Demo">

  <ext4:formPanel id="myFormPanel" title="Contact Info" width="800" height="300" bodyPadding="10">
    <ext4:textfield id="car" fieldLabel="Car" value="*" />

    <ext4:submitbutton text="Submit" nextPage="validateForm" addTo="secondPanel" replacePreviousPage="true" />

  </ext4:formPanel>

  <ext4:panel id="secondPanel" width="200" height="200" />

</ext4:container>
```

The first thing to notice is the *<ext4:formPanel>* tag which provides the special panel into which ExtJS formats the form fields.

Text Fields

In example above, we're using the simplest of the form fields, the *textfield*, by using the *<ext4:textfield>* tag. The *<ext4:textfield>* tag maps directly to the *Ext.form.field.Date* class and all of its Config Options can be used as attributes or child tags in the usual way.

If you're familiar with EWD's HTML form field handling, you'll see that it uses the *value="*"* convention for displaying the value of the EWD Session variable whose name matches the field's *id* value when the form is rendered. The default values for form fields are normally created in the page's *onBeforeRender* method, in this case *getFormData^Ext4Demo* which is as follows:

```
getFormData(sessid)
 d setSessionValue^%zewdAPI("car","Volvo",sessid)
 QUIT ""
 ;
```

This is all that is needed to set the initial default value for the *Car* text field.

Thirdly, the *<ext4:submitButton>* tag adds a Submit button. Note that it uses the by now familiar *nextPage/addTo* convention. In this example, pressing the submit button will fetch a fragment named *validateForm*. The *onBeforeRender* method of this fragment will be used to validate the values that were submitted.

The *validateForm.ewd* fragment is as follows:

```
<ext4:fragment onBeforeRender="checkForm^Ext4Demo">
  <ext4:panel html="#html" />
</ext4:fragment>
```

In this example, the validation fragment returns a panel, but it could be nothing more than a fragment with an onBeforeRender method if that was appropriate.

Its onBeforeRender method, checkForm^Ext4Demo, is as follows:

```
checkForm(sessid)
 d setSessionValue^%zewdAPI("html","Errors in form",sessid)
 n car
 s car=$$getSessionValue^%zewdAPI("car",sessid)
 ; the following is optional:
 d setFieldErrorAlert^%zewdExt4Code("Form Error!","You've made errors!",sessid)
 ; clear down any existing flagged errors:
 d clearFieldErrors^%zewdExt4Code(sessid)
 i car="" d setFieldError^%zewdExt4Code("car","You must enter a car name",sessid)
 i car="xxx" d setFieldError^%zewdExt4Code("car","I dont know that brand!",sessid)
 i '$$isFormErrors^%zewdExt4Code(sessid) d setSessionValue^%zewdAPI("html","Form validated OK",sessid)
 ; You must quit using this special function!
 QUIT $$formErrors^%zewdExt4Code(sessid)
```

There are a number of key features being used in the example above:

- by the time the *onBeforeRender* method has been invoked, the submitted value for the *car* textfield has been automatically put into the EWD Sessionby EWD. We can therefore get hold of the submitted value using the function *$$getSessionValue^%zewdAPI("car",sessid).*

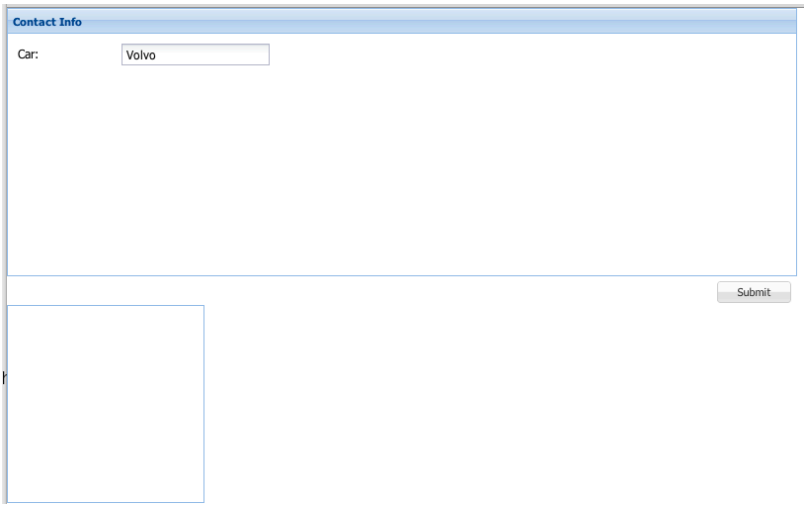
- Any validation errors will cause an ExtJS alert window to pop up. You can customise its contents using the *setFieldAlert^%zewdExt4Code()* method

- You denote any validation errors by invoking the *setFieldError^%zewdExt4Code()* method. This has the effect of highlighting the form field in question and adding a tooltip that shows the user the reason for the error when they hover over the field. You can flag as many errors as you wish within the form.

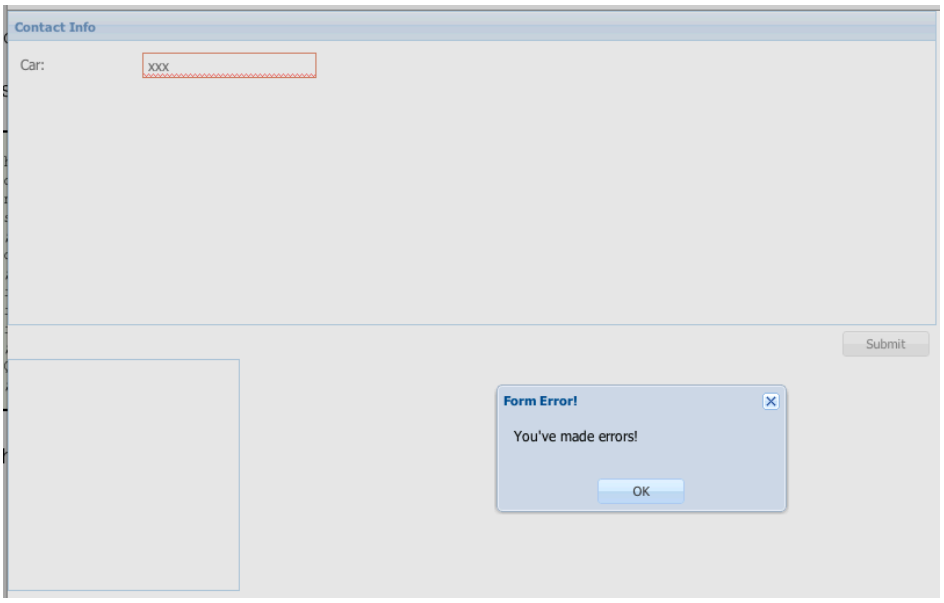
- If necessary, you can determine whether any errors have been found by using the *$$isFormErrors^%zewdExt4Code()* function.

- In order to correctly communicate back to the ExtJS form, you should QUIT with a *returnValue* that is provided by the *$$formErrors^%zewdExt4Code()* function. The actual *returnValue* that this produces depends on whether or not any validation errors were found by your validation checking logic.

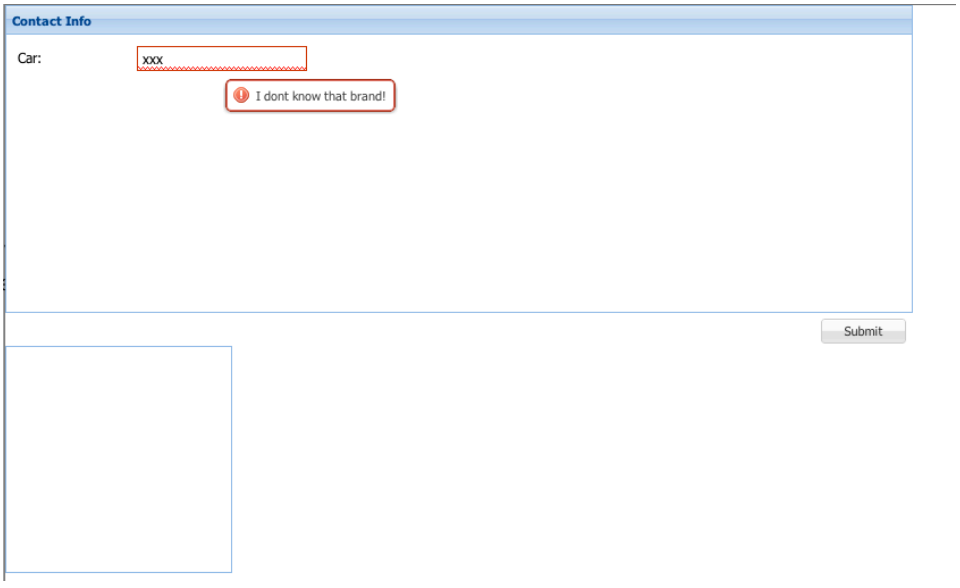
This example, when run, will look as follows when it first appears:



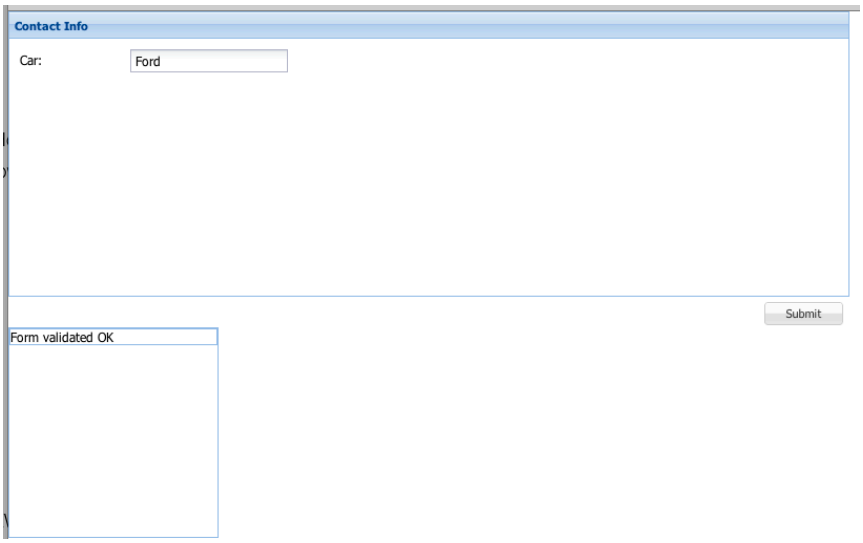
If you change the value of the *car* field to *xxx* and click the Submit button, you should get an alert:



If you click on the alert's OK button and hover the mouse pointer over the *car* field, you'll see the tooltip message we defined for this error:



However, if we change the car value to Ford and click Submit again, the fragment's panel will be sent and added to the lower panel:



You'll notice that we've included a validation check in the fragment's onBeforeRender method to flag an error if the value of car is an empty string. Although this is a reasonable thing to do, you might want to make use of a feature that is built-in to ExtJS: you can add the attribute *allowBlank="false"* to the textfield tag, ie:

```
<ext4:textfield id="car" fieldLabel="Car" value="*" allowBlank="false" />
```

You'll find that if you do this, you'll still be able to submit the form, so you'll still need to do the back-end validation to check for empty values. However the visible difference is that the tooltip will now be the ExtJS default one:

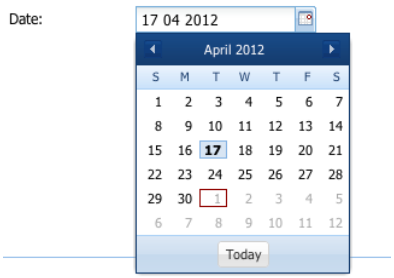
This field is required

As you can see, EWD provides a very simple and intuitive mechanism for integrating ExtJS forms, yet providing a sophisticated way of controlling their behaviour.

Let's now examine the other form field types. We'll add an instance of each one to our example form.

Date Fields

Date fields allow dates to be visually formatted in a wide variety of ways, and are edited using a calendar widget that automatically pops up, eg:



Here's an example of how to specify a date field:

```
<ext4:datefield id="myDate" fieldLabel="Date" value="*" submitFormat="m/d/Y" />
```

The *<ext4:datefield>* tag maps to the *Ext.form.field.Date* class and any of its Config Options can be used as attributes or child tags as appropriate.

If you want the dates presented to the user in a different format, use the format attribute, eg for UK format:

```
<ext4:datefield id="myDate" fieldLabel="Date" value="*" format="d/m/Y" submitFormat="m/d/Y" />
```

To pre-populate a date, use the same technique as for a textfield, but specify the date in US *m/d/Y* format (irrespective of the display format you want to use), eg:

```
d setSessionValue^%zewdAPI("myDate","04/20/2012",sessid)
```

This document provides details of the date format strings supported by ExtJS:

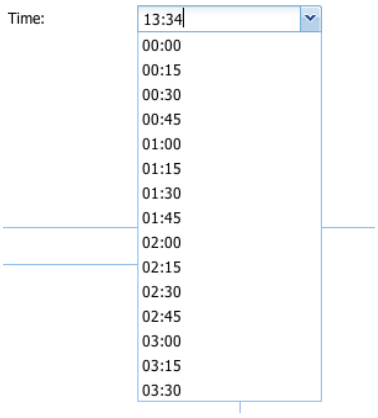
http://docs.sencha.com/ext-js/4-0/#!/api/Ext.Date

When the form is submitted, a date field will be submitted according to the submitFormat you've chosen. To use and/or validate the submitted value, get its value using the $$getSessionValue^%zewdAPI() method:

```
s date=$$getSessionValue^%zewdAPI("myDate",sessid)
```

Time Fields

Time fields allow times to be visually formatted in a wide variety of ways, and are edited using a time selector drop-down panel, eg:



Here's an example of how to specify a time field:

```
<ext4:timefield id="myTime" fieldLabel="Time" value="*" format="24Hour" minValue="08:00" maxValue="18:00"
   increment="15" />
```

The *<ext4:timefield>* tag maps to the *Ext.form.field.Time* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate a time, use the same technique as for a textfield, but specify the time according to the format you've specified in the tag, eg for the above example:

```
d setSessionValue^%zewdAPI("myTime","13:34",sessid)
```

This document provides details of the date and time format strings supported by ExtJS:

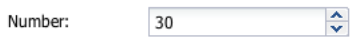
http://docs.sencha.com/ext-js/4-0/#!/api/Ext.Date

When the form is submitted, a time field will be submitted according to the format you've chosen. To use and/or validate it, get its value using the $$getSessionValue^%zewdAPI() method:

```
s time=$$getSessionValue^%zewdAPI("myTime",sessid) ; eg 09:30
```

Number Fields

Number fields allow numeric values to be visually formatted in a wide variety of ways, and are edited using a spinner widget, eg:



Here's an example of how to specify a number field:

```
<ext4:numberfield id="myNumber" fieldLabel="Number" value="*" minValue="10" maxValue="100" step="10" />
```

The *<ext4:numberfield>* tag maps to the *Ext.form.field.Number* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate a number field, use the same technique as for a textfield, eg:

```
d setSessionValue^%zewdAPI("myNumber",30,sessid)
```

To use and/or validate a submitted number field, get its value using the $$getSessionValue^%zewdAPI() method:

```
s number=$$getSessionValue^%zewdAPI("myNumber",sessid)
```

Display Fields

Display fields allow values to be displayed and presented within a form, but they cannot be edited, eg:

Here's an example of how to specify a display field:

```
<ext4:displayfield id="myDisplay" fieldLabel="Display" value="*" />
```

The *<ext4:displayfield>* tag maps to the *Ext.form.field.Display* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate a display field, use the same technique as for a textfield, eg:

```
d setSessionValue^%zewdAPI("myDisplay","Display Only text",sessid)
```

Hidden Fields

Hidden fields allow values to be added to a form, but they are not visible to and cannot be edited by the user, eg:

Here's an example of how to specify a hidden field:

```
<ext4:hiddenfield id="myHidden" value="*" />
```

The *<ext4:hiddenfield>* tag maps to the *Ext.form.field.Hidden* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate a hidden field, use the same technique as for a textfield, eg:

```
d setSessionValue^%zewdAPI("myHidden","This is a hidden value",sessid)
```

To use and/or validate a submitted hidden field, get its value using the $$getSessionValue^%zewdAPI() method:

```
s number=$$getSessionValue^%zewdAPI("myHidden",sessid)
```

Slider Fields

Slider fields allow numeric values to be visually formatted and edited using a slider widget, eg:

Here's an example of how to specify a number field:

```
<ext4:sliderfield id="mySlider" fieldLabel="Slider" width="250" value="*" increment="10" minValue="0"
    maxValue="100" />
```

The *<ext4:sliderfield>* tag maps to the *Ext.slider.Single* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate a slider field, use the same technique as for a textfield, eg:

```
d setSessionValue^%zewdAPI("mySlider",60,sessid)
```

To use and/or validate a submitted slider field value, get its value using the $$getSessionValue^%zewdAPI() method:

```
s value=$$getSessionValue^%zewdAPI("mySlider",sessid)
```

Radio Fields

Radio fields are the ExtJS version of radio buttons. They are normally grouped together to form a Radio Group, each radio field representing a possible, mutually exclusive value for the group, eg:

Here's an example of how to specify a group of radio fields:

```
<ext4:radiogroup fieldLabel="Two columns" columns="2" vertical="true">
  <ext4:radiofield boxLabel="Item 1" name="rb" inputValue="i1" />
  <ext4:radiofield boxLabel="Item 2" name="rb" inputValue="i2" />
  <ext4:radiofield boxLabel="Item 3" name="rb" inputValue="i3" />
  <ext4:radiofield boxLabel="Item 4" name="rb" inputValue="i4" />
  <ext4:radiofield boxLabel="Item 5" name="rb" inputValue="i5" />
  <ext4:radiofield boxLabel="Item 6" name="rb" inputValue="i6" />
</ext4:radiogroup>
```

The *<ext4:radiofield>* tag maps to the *Ext.form.Radio* class and any of its Config Options can be used as attributes or child tags as appropriate. Similarly, the *<ext4:radioGroup>* tag maps to the *Ext.form.RadioGroup* class.

To pre-check a radio field group, set the value of the correspondingly-named EWD Session value to match the inputValue of the radio button you want pre-checked. For example, the following will pre-check the Item 4 radio button in the example above:

```
d setSessionValue^%zewdAPI("rb","i4",sessid)
```

To use and/or validate the submitted response from a radio group, get the value of the radio button that was checked using the $$getSessionValue^%zewdAPI() method:

```
s valueChecked=$$getSessionValue^%zewdAPI("rb",sessid)
```

Checkbox Fields

Checkbox fields are the ExtJS version of checkboxes. They are normally grouped together to form a Checkbox Group, each checkbox field representing one possible value for the group, eg:

Here's an example of how to specify a group of checkbox fields:

```
<ext4:checkboxgroup fieldLabel="Two columns" columns="2" vertical="true">
  <ext4:checkboxfield boxLabel="Item 1" name="cb" inputValue="a1" />
  <ext4:checkboxfield boxLabel="Item 2" name="cb" inputValue="2b" />
  <ext4:checkboxfield boxLabel="Item 3" name="cb" inputValue="c3" />
  <ext4:checkboxfield boxLabel="Item 4" name="cb" inputValue="4" />
  <ext4:checkboxfield boxLabel="Item 5" name="cb" inputValue="5" />
  <ext4:checkboxfield boxLabel="Item 6" name="cb" inputValue="6" />
</ext4:checkboxgroup>
```

The *<ext4:checkboxfield>* tag maps to the *Ext.form.Checkbox* class and any of its Config Options can be used as attributes or child tags as appropriate. Similarly, the *<ext4:checkboxGroup>* tag maps to the *Ext.form.CheckboxGroup* class.

To pre-check a checkbox field group, use the standard EWD techniques for checkbox form fields: you must use one of two EWD APIs to create the EWD Selected Array:

- invoke the setCheckboxOn() method for each checkbox that should be checked

- create a local array of checkboxes to be checked, then invoke the setCheckboxValues() method.

For example, the following two alternatives will pre-check the first, third and sixth checkboxes in the example above:

```
d initialiseCheckbox^%zewdAPI("cb",sessid)
d setCheckboxOn^%zewdAPI("cb","a1",sessid)
d setCheckboxOn^%zewdAPI("cb","c3",sessid)
d setCheckboxOn^%zewdAPI("cb",6,sessid)
```

```
n sel
d initialiseCheckbox^%zewdAPI("cb",sessid)
s sel("a1")="a1"
s sel("c3")="c3"
s sel(6)=6
d setCheckboxValues^%zewdAPI("cb",.sel,sessid)
```

The *initialiseCheckbox()* method clears any previously checked values in the EWD Session Array.

To use and/or validate the submitted responses from a checkbox group, you extract the values from the EWD Selected Array, again using the standard EWD APIs for checkboxes. Again, there are two alternative techniques.

To determine whether a specific checkbox was checked when the form was submitted:

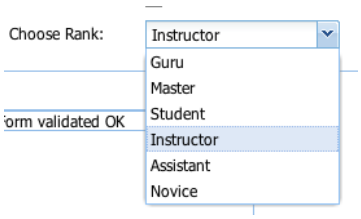
```
i $$isCheckboxOn^%zewdAPI("cb","c3",sessid) s result="Third checkbox was selected"
```

To create a local array of checked fields in the submitted checkbox group:

```
d getCheckboxValues^%zewdAPI("cb",.selected,sessid)
; format of array: selected("c3") = "c3"
```

Combobox Fields

The ExtJS Combobox field provides a more sophisticated alternative to the HTML <select> field, and can act as a proper combobox widget, eg:



Here's an example of how to specify a combobox field:

```
<ext4:comboBox fieldLabel="Choose Rank" name="rank" value="*" />
```

The *<ext4:comboboxfield>* tag maps to the *Ext.form.field.Combobox* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate a combobox field, use the standard EWD techniques for an HTML <select> tag. There are two steps to this:

- defining the list of possible options: use the standard *appendToList()* method

- specifying the option to be pre-highlighted: use the *setSessionValue()* method

For example:

```
d clearList^%zewdAPI("rank",sessid)
d appendToList^%zewdAPI("rank","Guru","guru",sessid)
d appendToList^%zewdAPI("rank","Master","master",sessid)
d appendToList^%zewdAPI("rank","Student","student",sessid)
d appendToList^%zewdAPI("rank","Instructor","teacher",sessid)
d appendToList^%zewdAPI("rank","Assistant","aid",sessid)
d appendToList^%zewdAPI("rank","Novice","novice",sessid)
d setSessionValue^%zewdAPI("rank","student",sessid)
```

The *clearList()* method is used as a safeguard, to clear down any values that might already exist in the *rank* Session List Array.

To use and/or validate a submitted combobox field value, simply get its value using the $$getSessionValue^%zewdAPI() method:

```
s value=$$getSessionValue^%zewdAPI("rank",sessid)
```

MultiSelect Combobox Field

You can optionally specify that multiple values can be selected from a combobox list. To do this, use the multiSelect="true" attribute and remove the value="*" attribute, eg:

```
<ext4:comboBox fieldLabel="Choose Rank" name="rank" multiSelect="true" />
```

To specify the values to be pre-selected, use the same EWD technique as you'd use for a <select multiple> tag, eg:

```
d initialiseMultipleSelect^%zewdAPI("rank",sessid)
d setMultipleSelectOn^%zewdAPI("rank","student",sessid)
d setMultipleSelectOn^%zewdAPI("rank","aid",sessid)
```

When the form is submitted, you can determine which value(s) were selected by using the standard EWD multipleSelect techniques, eg:

```
i $$isMultipleSelectOn^%zewdAPI("rank","teacher",sessid) s result="Instructor was selected"
```

To create a local array of selected fields in the submitted combobox field:

```
d getMultipleSelectValues^%zewdAPI("rank",.selected,sessid)
; format of array: selected("student") = "student"
```

Textarea Fields

Textarea fields are the ExtJS version of the HTML *textarea*, allowing multi-line text blocks to be created and edited using a simple text editor interface, eg:

Here's an example of how to specify a textarea field:

```
<ext4:textareafield grow="true" name="message" fieldLabel="Message" anchor="100%" value="*" />
```

Note the inclusion of the value="*" attribute which is essential if you want default text to be displayed when the field is rendered.

The *<ext4:textareafield>* tag maps to the *Ext.form.field.TextArea* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate a *textarea* field, use the special EWD ExtJS-specific API for *textarea* form fields:

setTextAreaValue^%zewdExt4Code()

For example, the following *onBeforeRender* method code will pre-populate the *textarea* field in the example above:

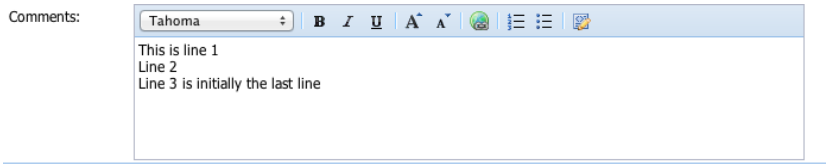
```
n mess
s mess(1)="This is line 1"
s mess(2)="Line 2"
s mess(3)="Line 3 is initially the last line"
d setTextAreaValue^%zewdExt4Code(.mess,"message",sessid)
```

To use and/or validate the submitted response from a *textarea* field, you extract the values from the EWD TextArea Session Array, again using the standard EWD API:

```
 n array
 d getTextArea^%zewdAPI("message",.array,sessid)
 ; array format:
 ;  array(0) = number of lines of text
 ;  array(lineNumber) = line of text
```

HTMLEditor Fields

The HTML Editor field is a substitute for the basic *textarea* field, allowing multi-line text blocks to be created and edited using a rich text editor interface, eg:



Here's an example of how to specify an HTML Editor field:

```
<ext4:htmleditorfield name="comments" fieldLabel="Comments" enableColors="false" enableAlignments="false"
   value="*" />
```

Note the inclusion of the value="*" attribute which is essential if you want default text to be displayed when the field is rendered.

The *<ext4:htmleditorfield>* tag maps to the *Ext.form.field.HtmlEditor* class and any of its Config Options can be used as attributes or child tags as appropriate.

To pre-populate an *HtmlEditor* field from a simple text array, use the special EWD ExtJS-specific API:

setHtmlEditorValue^%zewdExt4Code()

For example, the following onBeforeRender method code will pre-populate the *HtmlEdiitor* field in the example above:

```
n mess
s mess(1)="This is line 1"
s mess(2)="<i>Line 2</i>"
s mess(3)="Line 3 is initially the last line"
d setHtmlEditorValue^%zewdExt4Code(.mess,"comments",sessid)
```

However, if the text is relatively short, you can simply use *setSessionValue()*, eg:

```
s text="This is line 1<br><i>Line 2</i><br>Line 3 is initially the last line<br><br>new last line!"
d setSessionValue^%zewdAPI("comments",text,sessid)
```

Note that the highlighting is done using standard HTML tags.

When submitted, the contents of an *HtmlEditor* field is copied to the EWD Session as usual. If the content is short, it will be copied to a simple scalar EWD Session Value. However, if the text is long, you may find that EWD has automatically broken it up and merged it into an EWD Session Array. *Note however that this automatic record splitting will not necessarily have occurred at line boundaries.*

Hence for short text, you can access it using:

```
s comments=$$getSessionValue("comments",sessid)
```

But large amounts of text will be accessed using:

```
n comments
d $$mergeArrayFromSession(.comments,"comments",sessid)
```

The ExtJS Desktop

Demystifying the ExtJS Desktop

The ExtJS examples includes an amazingly cool concept called the ExtJS Desktop which emulates a Windows-style desktop UI running within a browser. Strangely, there is no documentation readily available about how to build your own equivalent UI.

However, you'll find that by using EWD you can make use of this truly ground-breaking concept, and in a matter of a few minutes you'll have the basic outline of a web-based desktop that you can then customise to behave exactly how you wish, and fully integrated with your Caché or GT.M back-end database.

A Simple Desktop

So let's create a simple desktop application. First create a container page as follows:

```
<ext4:container rootPath="/ext-4">

  <ext4:desktop>
    <ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
      fragment="dtWin1" />
  </ext4:desktop>

</ext4:container>
```

So what we've done is to specify that this container page is a desktop by using the *<ext4:desktop>* tag. This should be the only immediate child tag of the *<ext4:container>* tag. Inside the *<ext4:desktop>* tag, we've defined a single *<ext4:window>* tag. This has two effects:

- it defines a desktop icon

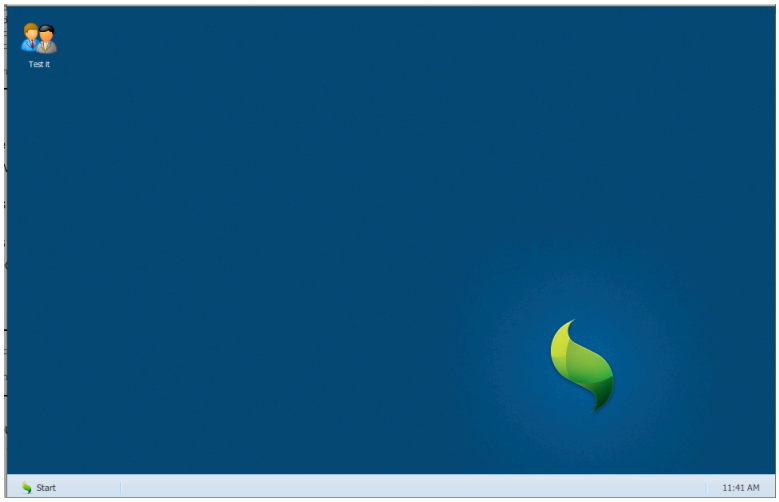
- if defines an initially empty window that will pop up when the icon is clicked. The height and width attributes define the dimensions of the window. This window is then automatically populated by the fragment specified in the *fragment* attribute.

So, let's create a simple fragment to go into the window. Here's the initial contents of *dtWin.ewd*:

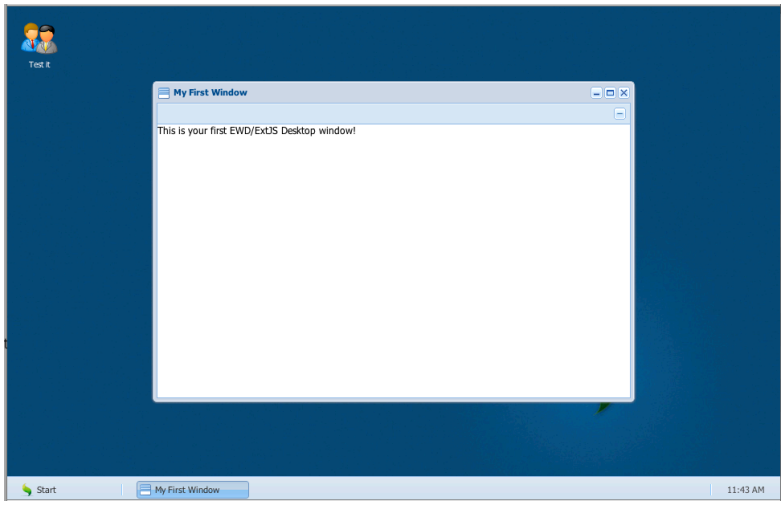
```
<ext4:fragment>
  <ext4:panel html="This is your first EWD/ExtJS Desktop window!" border="0" />
</ext4:fragment>
```

OK that's all you need to create! Now compile these two EWD pages and start the container page in your browser. Don't worry if the compilation for the container page takes longer than usual: it has a lot of work to do!

You should see the following:

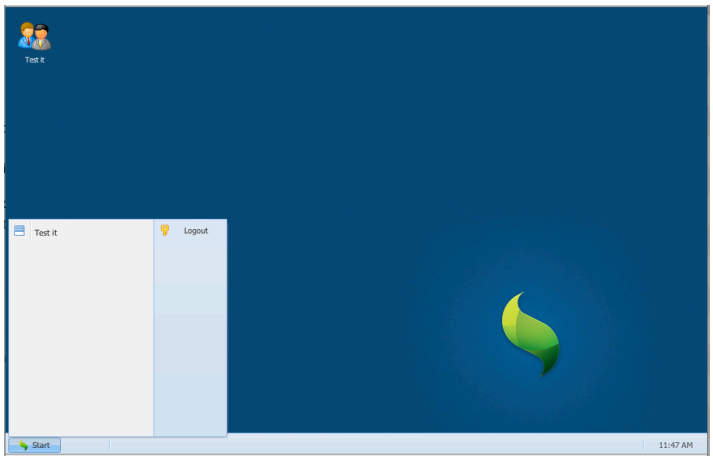


Try clicking on the "Test it" icon and the window should pop up:



Just like a Windows UI, you'll see an icon for the window on the bottom status bar. You'll find that you can move the window, resize it and minimise it, just like a standard Windows window. Clicking on the bottom status bar icon will return it to view. The window can be closed by clicking the X in the top right corner of the window.

Close the window and click the Start icon at the bottom left of the browser. Again, just like in Windows, a Start panel will pop up, listing the icons on the desktop:



You can open your test window by clicking on the "Test it" option in the Start panel.

The Logout Option

You'll see a *Logout* option in the Start panel. If you click it, an alert will pop up telling you *"undefined logout function"*. That's because we haven't told the desktop how to handle it. So let's add a handler to the container page as follows:

```
<ext4:container rootPath="/ext-4">

  <ext4:desktop logoutFn="function() {alert('logging out!');}">
    <ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
      fragment="dtWin1" />
  </ext4:desktop>

</ext4:container>
```

Recompile the container page and re-run it, then try clicking the *Logout* icon. It should now bring up your alert instead of the default one. So now let's create a typical logout mechanism. First create another fragment named *logout.ewd*:

```
<ext4:fragment>
 <ext4:js at="top">
  document.location.replace('/loggedout.html');
 </ext4:js>
</ext4:fragment>
```

Next, create a standard HTML file named *loggedout.html* that you're going to save into your web-server's root path:

```
<html>
 <head>
  <title>Logged Out</title>
 </head>
 <body>
  <h1>You have now been logged out!</h1>
 </body>
</html>
```

This page isn't very pretty: you can style it up later if you wish. However, it's enough for this simple demo. Finally edit your logout handler function so that it fetches the *logout* fragment:

```
<ext4:container rootPath="/ext-4">

  <ext4:desktop logoutFn="function() {EWD.ajax.getPage({page:'logout'});}">
    <ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
      fragment="dtWin1" />
  </ext4:desktop>

</ext4:container>
```

Recompile these pages and restart the desktop application. Now when you click the *logout* option in the Start panel, you'll be redirected to the *loggedout.html* page. By using the *document.location.replace()* function, you'll find that you can't use the back button to return to the desktop, which is the safest technique to use for logging out a user.

Defining a UserName

You can define a user name that will appear in the desktop Start Panel. You do this in the Container Page's onBeforeRender method:

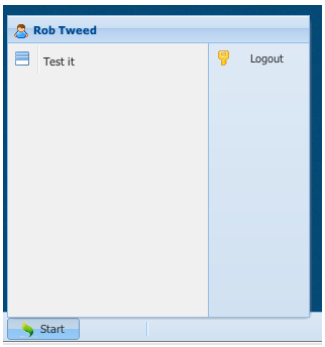
```
<ext4:container rootPath="/ext-4" onBeforeRender="setusername^Ext4Demo">

  <ext4:desktop logoutFn="function() {EWD.ajax.getPage({page:'logout'});}">
    <ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
      fragment="dtWin1" />
  </ext4:desktop>

</ext4:container>
```

This *onBeforeRender* method will be something like this:

```
setusername(sessid)
 d setSessionValue^%zewdAPI("EWD.desktop.username","Rob Tweed",sessid)
 QUIT ""
 ;
```

EWD.desktop.username is a reserved EWD Session Name.

You'll now find that the Start Panel shows this username:



Adding a Login Mechanism

So let's look at how you could build an initial login dialog that would validate the user's credentials and bring up the desktop showing them logged in with their username. You'd need to do this using two separate container pages, one for a login dialogue, which, if successful, switches to the desktop container page.

First create a simple login form page, using a modal window. We'll put this inside a viewport container so that it occupies all the available browser space. Create a container page named *login.ewd* as follows:

```
<ext4:container rootPath="/ext-4">
  <ext4:viewPort layout="fit">
    <ext4:modalwindow title="Please Log In" height="200" width="400" layout="fit" autoShow="true">
      <ext4:formPanel bodyPadding="10">
        <ext4:textfield id="username" fieldLabel="Username:" allowBlank="false" value="" />
        <ext4:textfield id="password" inputType="password" fieldLabel="Password:" allowBlank="false" value="" />
        <ext4:submitbutton text="Submit" nextPage="loginRedirect" />
      </ext4:formPanel>
    </ext4:modalwindow>
  </ext4:viewPort>
</ext4:container>
```

Clicking the submit button will fetch a fragment named *loginRedirect.ewd*. It should contain the following:

```
<ewd:fragment onBeforeRender="login^Ext4Demo">
 <script language="javascript">
  document.location.replace('desktop.ewd');
 </script>
</ewd:fragment>
```

The verification of the user's credentials is performed by this fragment's *onBeforeRender* method which contains:

```
login(sessid)
 n username,password
 d clearFieldErrors^%zewdExt4Code(sessid)
 s username=$$getSessionValue^%zewdAPI("username",sessid)
 s password=$$getSessionValue^%zewdAPI("password",sessid)
 i username="" d setFieldError^%zewdExt4Code("username","You must enter a username",sessid)
 i password="" d setFieldError^%zewdExt4Code("password","You must enter a password",sessid)
 i username'="rob" d setFieldError^%zewdExt4Code("username","Unrecognised username",sessid)
 i password'="1234" d setFieldError^%zewdExt4Code("password","Invalid password",sessid)
 QUIT $$formErrors^%zewdExt4Code(sessid)
 ;
```

This is hard-coded to only allow a username/password combination of *rob/1234*. You can modify it to use your existing login validation mechanism.

The final change that is necessary is to reset the *desktop.ewd* page to no longer be a *FirstPage*: ie it cannot be loaded with an untokenised URL, so it can only be started after a successful login:

```
<ext4:container isFirstPage="false" rootPath="/ext-4" onBeforeRender="setusername^Ext4Demo">

  <ext4:desktop logoutFn="function() {EWD.ajax.getPage({page:'logout'});}">
    <ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
      fragment="dtWin1" />
  </ext4:desktop>

</ext4:container>
```

Try it out: you now have a completely secured desktop application with a login and logout mechanism.

Customising & Extending the Desktop

Window Contents

Since the windows that open are standard ExtJS window containers, you can populate them with any other ExtJS components. What you put into the windows is entirely up to you: just put whatever combination of other ExtJS components you want into the fragment that you specify for each window. Remember that this fragment can, in turn, fetch others in the standard way described in this document. The one thing you must be careful of, however, is to ensure that at any one time, only one instance of a particular id is present in the Container Page. ExtJS can get very confused if you have more than one component with the same id.

Positioning Icons

You can add as many icons/windows as you like to the desktop. By default they will be added in a column on the left-hand side, but you'll find that eventually you'll run out of room. You can place icons anywhere on the desktop yourself by adding the *position="absolute"*, *left* and *top* attributes, eg:

```
<ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
     fragment="dtWin1" position="absolute" left="200" top="100" />
```

Quickstart Icon

You can add a *quickstart* icon on the desktop's bottom status bar by adding the *quickstart="true"* attribute, eg:

```
<ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
     fragment="dtWin1" quickstart="true" />
```

Desktop Icons

By default, the desktop icon uses the ExtJS “accordion-shortcut” CSS class to define it. You can use other icons by changing this class: use the iconCls attribute, eg:

```
<ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
     fragment="dtWin1" iconCls="cpu-shortcut" />
```

ExtJS provides a number of built-in classes you can try out:

- accordion-shortcut
- cpu-shortcut
- grid-shortcut
- notepad-shortcut

You could add your own, if required, by editing the */examples/desktop/css/desktop.css* file that you'll find in your ExtJS distribution filesystem.

Window Icons

By default, the small icons within the window banners and in the Start panel use the ExtJS “accordion” CSS class to define them. You can use other icons by changing this class: use the windowIconCls attribute, eg:

```
<ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
     fragment="dtWin1" windowIconCls="icon-grid" />
```

ExtJS provides a number of built-in classes you can try out:

- accordion
- icon-grid
- tabs

You could add your own, if required, by editing the */examples/desktop/css/desktop.css* file that you'll find in your ExtJS distribution filesystem.

Desktop Wallpaper

The default wallpaper uses the ExtJS-provided file:

examples/desktop/wallpapers/Blue-Sencha.jpg

You can use a different wallpaper file by adding the wallpaper attribute to the <ext4:desktop> tag, eg:

```
<ext4:container isFirstPage="false" rootPath="/ext-4" onBeforeRender="setusername^Ext4Demo">

  <ext4:desktop logoutFn="function() {EWD.ajax.getPage({page:'logout'});}"
       wallpaper="/ext-4/examples/desktop/wallpapers/desk.jpg">
    <ext4:window title="My First Window" name="Test it" id="testIcon" width="600" height="400"
      fragment="dtWin1" />
  </ext4:desktop>

</ext4:container>
```

You'll find a variety of ExtJS-defined wallpapers in the */examples/desktop/wallpapers* folder.

Dynamically-defined Desktop

The previous examples have used explicitly-defined tags to describe the desktop. It is often desirable to make the desktop completely dynamically-defined. For example, you may want to adjust the range of icons that are put on the desktop according to the user's profile. EWD allows you to dynamically define all aspects of a desktop in much the same way as Grids, by using a simple EWD Session Array.

To create a dynamically-defined desktop, simply provide the *<ext4:desktop>* tag and add the *sessionName* attribute, eg:

```
<ext4:container isFirstPage="false" rootPath="/ext-4" onBeforeRender="setusername^Ext4Demo">

  <ext4:desktop sessionName="desktop" />

</ext4:container>
```

The onBeforeRender method must now be extended to define the desktop and its properties, eg:

```
desktopConfig2(sessid)
 n desktop
 ;
 s desktop("username")="Rob Tweed"
 s desktop("wallpaper")="/ext-4/examples/desktop/wallpapers/desk.jpg"
 ;
 s desktop("logoutFn")="function() {alert('logging you out!');}"
 s desktop("windows",1,"title")="Simpsons Grid"
 s desktop("windows",1,"name")="Simpsons"
 s desktop("windows",1,"iconCls")="accordion-shortcut"
 s desktop("windows",1,"id")="myWin1"
 s desktop("windows",1,"width")=500
 s desktop("windows",1,"height")=350
 s desktop("windows",1,"fragment")="fragtest"
 s desktop("windows",1,"quickStart")="true"
 ;
 s desktop("windows",2,"title")="My 2nd window"
 s desktop("windows",2,"name")="Second Window"
 s desktop("windows",2,"iconCls")="accordion-shortcut"
 s desktop("windows",2,"windowIconCls")="accordion"
 s desktop("windows",2,"id")="myWin2"
 s desktop("windows",2,"width")=550
 s desktop("windows",2,"height")=400
 s desktop("windows",2,"fragment")="dtWin1"
 s desktop("windows",2,"position")="absolute"
 s desktop("windows",2,"left")=200
 s desktop("windows",2,"top")=100
 s desktop("windows",2,"quickStart")="true"
 ;
 d mergeArrayToSession^%zewdAPI(.desktop,"desktop",sessid)
 QUIT ""
```

Hopefully the mapping between this Session Array-based description of the desktop and the explicitly-defined tab-based version described previously is intuitive and obvious. Try experimenting with the properties and their values. You'll now have a desktop container page that only needs to be compiled once. Every time it runs, the actual characteristics and content of the desktop will now depend on the EWD Session Array contents that are generated at run-time!

You now have at your disposal, one of the most powerful web application frameworks in the world: the combination of ExtJS v4 and EWD!

Appendix 1: Installing EWD

Installing EWD

GT.M

The quickest and simplest way to use EWD and GT.M is to download and use the dEWDrop Virtual Machine (*http://www.fourthwatchsoftware.com)* which is pre-configured with the latest build of EWD. However, if you wish to build your own system, you can get the latest EWD routine files from *https://github.com/robtweed/EWD.* See our website *(http://www.mgateway.com/ewd.html*) for details on installing EWD on GT.M systems.

Caché

You should download a copy of the latest version of EWD from our web site *(http://www.mgateway.com)*:

- Click the ***Enterprise Web Developer*** tab
- Click the tabs ***Download EWD*** followed by ***EWD for Caché***.
- Complete the registration form and you'll be able to download the latest copy of EWD for free. The Sencha Touch custom tags are included in EWD.

The zip file that you'll download contains one critical file:

- **zewd.xml** - the object code file that you install into your *%SYS* namespace using *$system.OBJ.Load()*. Let this overwrite any existing copy of *^%zewd** routines if you already have EWD on your Caché system

Configuring EWD

EWD can generate CSP, WebLink and GT.M versions of Mobile web applications from the same EWD application source code. If you're already using EWD, then you can immediately start developing EWD applications.

If you're new to EWD, then you'll need to configure EWD for either WebLink, CSP or GT.M, depending on which technology you use. There are configuration instructions on our web site, but here's a quick way of configuring them, based on certain assumptions - just change the references according to your exact GT.M or Caché/WebLink/CSP configuration.

Caché & CSP

1a) Simple Default Configuration

If you are using a default Caché installation and want to initially use the built-in Apache web server that is configured to use port 57772, you can just run (in a Caché Terminal session):

```
do configureDefault^%zewdCSP
```

This sets up the configuration global ^zewd for you.

1b) Custom Configuration

However, if you have configured IIS or some other web server for use with CSP, you'll need to manually configure EWD as appropriate to your specific configuration. This is done via the global *^zewd*. Here's an example of how to do this:

Assumptions:

- you'll be running your EWD-generated CSP applications in your *USER* namespace
- you're using IIS as your web server and its root path is *c:\inetpub\wwwroot*
- your source EWD applications will reside under the path *c:\ewdapps*
- the CSP application directories and files generated by EWD will be saved under *c:\InterSystems\Caché\CSP\ewd*

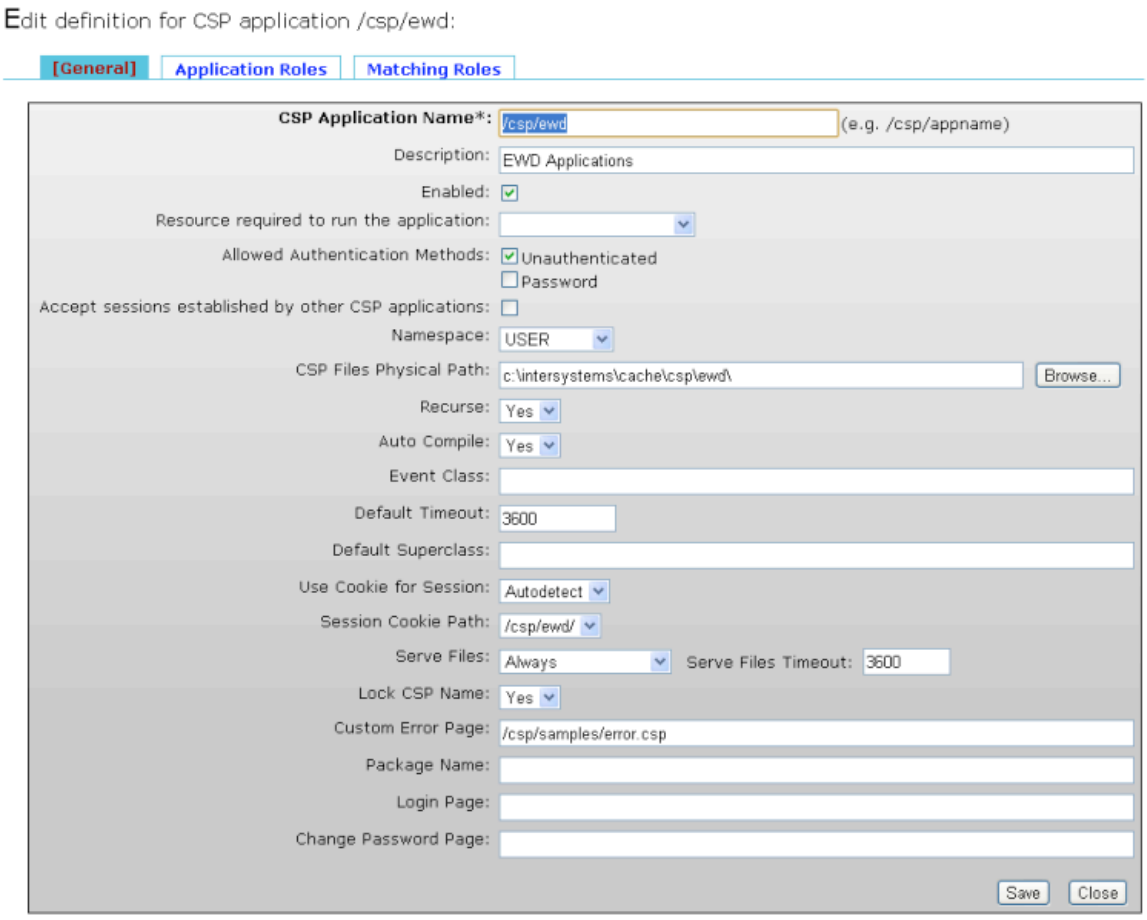
Create a global named *^zewd* as follows (adjust as necessary):

```
^zewd("config","RootURL","csp")="/csp/ewd"
^zewd("config","applicationRootPath")="c:\ewdapps"
^zewd("config","outputRootPath","csp")="c:\InterSystems\Cache\CSP\ewd"
^zewd("config","jsScriptPath","csp","mode")="fixed"
^zewd("config","jsScriptPath","csp","path")="/"
^zewd("config","jsScriptPath","csp","outputPath")="c:\Inetpub\wwwroot"
```

2) Define CSP Application

Next, you must create a CSP Application named *"/csp/ewd"* that points to the *outputRootPath* above and directs you to the required namespace (*USER*). To so this, use the *Caché System Management Portal*, select *Security Management/ CSP Applications*, then click the *Create New CSP Application* link.

Fill out the form as shown below to get you started:



The settings shown above are for a simple default CSP system using the built-in web server. If you have a customized CSP configuration, you may need to make some adjustments, in particular to the CSP Files Physical Path.

EWD should now be ready to use with CSP.

Caché & WebLink

Assumptions:

- you'll be running your EWD applications in your USER namespace
- you're using IIS as your web server and its root path is *c:\inetpub\wwwroot*
- your source EWD applications will reside under the path *c:\ewdapps*
- you'll be using the WebLink Server (*MGWLPN*) *LOCAL* which, by default, connects incoming requests to the *USER* namespace

Create a global named ^zewd in the USER namespace as follows (adjust as necessary):

```
^zewd("config","RootURL","wl")="/scripts/mgwms32.dll"
^zewd("config","applicationRootPath")="/usr/ewdApps"
^zewd("config","jsScriptPath","wl")="fixed"
^zewd("config","jsScriptPath","wl","mode")="fixed"
^zewd("config","jsScriptPath","wl","outputPath")="c:\Inetpub\wwwroot"
^zewd("config","jsScriptPath","wl","path")="/"
```

You also must create the global (again in *USER*):

```
^MGWAPP("ewdwl")="runPage^%zewdWLD"
```

This latter global creates the WebLink dispatcher to EWD's WebLink run-time engine.

GT.M

Examples assume that you are running a GT.M and EWD configuration, defined as per the dEWDrop Virtual machine:

- you'll be in the */home/vista/* path when you start the GT.M shell using *mumps -dir*
- you're using Apache as your web server and its root path is */home/vista/www*
- your source EWD applications will reside under the path */home/vista/www/ewd*
- *m_apache* has been installed and configured to dispatch to EWD's runtime code when URLs are encountered containing */vista*
- Javascript and CSS files that are generated by EWD will be saved under the webserver path */vista/resources*

Create a global named ^zewd as follows (adjust as necessary):

```
^zewd("config","RootURL","gtm")="/vista/"
^zewd("config","applicationRootPath")="/home/vista/www/ewd"
^zewd("config","jsScriptPath","gtm","mode")="fixed"
^zewd("config","jsScriptPath","gtm","outputPath")="/home/vista/www/resources/"
^zewd("config","jsScriptPath","gtm","path")="/vista/resources/"
^zewd("config","routinePath","gtm")="/home/vista/www/r/"
```

Creating EWD Pages

This tutorial will guide you through the process, but here's a quick summary of the process involved, based on the configuration settings shown above.

Having configured your EWD environment, you should now be ready to start developing. Create your new EWD application source pages in subdirectories of the Application Root Path, *eg* if your Application Root Path is *c:\ewdapps* and your application is named *myApp*::

```
c:\ewdapps\myApp\index.ewd

c:\ewdapps\myApp\login.ewd
```

You can use any text editor to create and edit these files.

To create an executable web application from these pages, you must compile them. This is most easily done using the command-line APIs that you invoke from within Caché Terminal or, if you are using GT.M, from within a Linux terminal session running the GT.M shell.

To compile an entire application (eg one named myApp):

CSP:

```
USER> d compileAll^%zewdAPI("myApp",,"csp")
```

WebLink:

```
USER> d compileAll^%zewdAPI("myApp",,"wl")
```

GT.M:

```
USER> d compileAll^%zewdAPI("myApp")
```

To compile one page (*eg myPage.ewd*) in an application (eg *myApp*):

CSP:

```
USER> d compilePage^%zewdAPI("myApp","myPage",,"csp")
```

WebLink:

```
USER> d compilePage^%zewdAPI("myApp","myPage",,"wl")
```

GT.M:

```
USER> d compilePage^%zewdAPI("myApp","myPage")
```

Running EWD Applications

You'll now have a runnable Web Application that will run in a desktop browser. The structure of the URL you'll use to invoke and start the application depends on whether you're using GT.M, WebLink or CSP:

CSP

For CSP EWD applications, the structure of the URL you'll use is:

http://127.0.0.1/csp/ewd/[applicationName]/[pageName].csp

where: **applicationName** is the name of your EWD application

pageName is the name of the first page of your EWD application

for example:

http://127.0.0.1/csp/ewd/myApp/index.csp

WebLink

For WebLink EWD applications, the structure of the URL you'll use is:

http://127.0.0.1/scripts/mgwms32.dll?
MGWLPN=LOCAL&MGWAPP=ewdwl&app=[applicationName]&page=[pageName]

where **applicationName** is the name of your EWD application

pageName is the name of the first page of your EWD application

for example:

http://127.0.0.1/scripts/mgwms32.dll?MGWLPN=LOCAL&MGWAPP=ewdwl&app=myApp&page=index

If you're using Apache, you'll typically replace ***/scripts/mgwms32.dll*** with ***cgi-bin/nph-mgwcgi***

Of course if you're using a WebLink Server other than *LOCAL*, you'll also need to change the value of the *MGWLPN* name/value pair.

GT.M

For GT.M EWD applications, the structure of the URL you'll use is:

http://127.0.0.1/vista/[applicationName]/[pageName].ewd

where **applicationName** is the name of your EWD application

pageName is the name of the first page of your EWD application

for example:

http://127.0.0.1/vista/myApp/index.ewd