



EWD SMART Container

Reference Guide

Table of Contents

Introduction	1
Background	1
Installation and Configuration	2
Installing EWD's SMART Container	2
The SMART Application	2
The ontology.xml File	3
The smartVistADemo Application	3
Maintaining SMART Manifests	4
About SMART Manifests	4
Registering Manifests on an EWD System	4
Creating a SMART Container	8
The Basics of a Vista SMART Container	8
The Login Form	8
The Main Layout Page	10
Patient Selection	11
Running a SMART App via its Manifest	14
Appendix 1: Installing EWD	17
Installing EWD	17
GT.M	17
Caché	17

Configuring EWD	17
Caché & CSP	18
1a) <i>Simple Default Configuration</i>	18
1b) <i>Custom Configuration</i>	18
2) <i>Define CSP Application</i>	18
Caché & WebLink	19
GT.M	20
Creating EWD Pages	20
Running EWD Applications	21

Introduction

Background

Enterprise Web Developer (EWD) is a framework for rapidly building web applications that integrate with the Caché and GT.M databases. EWD automates many necessary processes that must be present in web applications, including security management, state management and session management.

SMART (Substitutable Medical Apps, Re-usable Technology) is an initiative from Harvard Medical School (<http://smartplatforms.org>) that provides a unified mechanism for diverse applications to interact with medical record data. SMART Applications (Apps) built against the SMART API can be embedded within any SMART Container. A SMART Container is, most often, an Electronic Medical Record (EMR) system used by physicians, but might also be a Personal Health Record (PHR) used by patients, or a data-analytics platform used by researchers. A SMART App is a normal web application, embedded as a frame within the SMART Container's user interface, with access to the SMART API for interacting with health data.

EWD includes an integrated SMART Container capability. EWD's SMART Container functionality has been implemented primarily for use with the US Department of Veterans Affairs EMR known as VistA, but can be used with any Caché or GT.M-based EMR or PMR.

This document explains how to install and use EWD's SMART Container. The examples are based on VistA, but guidance is included on how the SMART Container can be used with other platforms.

Installation and Configuration

Installing EWD's SMART Container

The SMART Container functionality is incorporated into EWD Build 939 and later. If you are already using EWD and want to use the SMART Container, you should upgrade to Build 939 or later:

- Caché users can find the latest build of EWD at <http://www.mgateway.com/ewdDownload.html> - click the EWD for Caché tab and follow the instructions to download it
- GT.M users can find the latest Open Source build of EWD at <https://github.com/robtweed/EWD>

For readers who are new to EWD, Appendix I in this document provides details on how to install it.

You'll need to install the ExtJS Javascript framework in order to use some of EWD's support & management tools for SMART Apps. For details on ExtJS and how to use it with EWD, see http://gradvs1.mgateway.com/download/EWD_EXTJS4_Reference.pdf, particularly the chapter on Installation and Configuration at page 9.

The SMART Application

A key part of EWD's SMART Container is an EWD Application named *SMART*. The source EWD pages for this application can be found at and downloaded from <https://github.com/robtweed/EWD/tree/master/SMART/ewdapps/SMART>

If you haven't already done so, you should create a new EWD Application directory named *SMART*. This should be created as a sub-directory of your EWD Application Root Path (eg on a dEWDrop system the EWD Application Root Path is */home/vista/www/ewd* and you will find that the directory */home/vista/www/ewd/SMART* has already been created). Copy the EWD pages that you downloaded from the Github repository into the new *SMART* directory (on a dEWDrop system you'll find that these EWD pages have been pre-installed).

The SMART Application contains a number of sub-components including:

- a web-based sub-system that you should use to manage and maintain the SMART App manifests that you need in order to run SMART Apps;
- a pre-built, EWD-aware SMART Container frame that will be automatically created for you when running SMART Apps
- a special EWD page that is automatically used when you use REST-based SMART Apps

Note that the *SMART* Application makes use of the ExtJS v4 Javascript framework, so you must ensure that you have installed this. For details on installing ExtJS v4 for use with EWD, see http://gradvs1.mgateway.com/download/EWD_EXTJS4_Reference.pdf (Note: ExtJS v4 is already pre-installed on dEWDrop systems).

The back-end onBeforeRender scripts for the SMART application pages are all included in the routine `^%zewdSmart` (see https://github.com/robtweed/EWD/blob/master/_zewdSmart.m Note: an Apache-licensed copy of the same source code can be found at https://github.com/robtweed/EWD/blob/master/SMART/_zewdSmart.m)

Ensure that you compile the SMART EWD Application (it is already pre-compiled on dEWDrop systems).

The ontology.xml File

Another key component of the SMART Container is the *ontology.xml* file. You'll find this in the Github repository at <https://github.com/robtweed/EWD/tree/master/SMART/ontology>

You should copy this file into the same directory as the SMART EWD pages (as described in the previous section). For example, on a dEWDrop system, you'll find that the file */home/vista/www/ewd/SMART/ontology.xml* already exists.

The smartVistADemo Application

The EWD Github repository also includes a demonstration SMART Container application named *smartVistADemo*.

This application can be used as-is to provide a convenient, ready-made platform for running SMART Apps on a Vista system, but is also provided as a fully-worked example, complete with full source code, that can be used as the basis of your own custom SMART Container applications.

To install the *smartVistADemo* application, download the pages from <https://github.com/robtweed/EWD/tree/master/SMART/ewdapps/smartVistADemo> (On dEWDrop systems you will find this application already pre-installed in the directory */home/vista/www/ewd/smartVistADemo*)

Ensure that you compile the *smartVistADemo* EWD Application (it is already pre-compiled on dEWDrop systems).

The back-end *onBeforeRender* method scripts for the *smartVistADemo* application are included in the *smartVistADemo.m* file that you'll also find in the <https://github.com/robtweed/EWD/tree/master/SMART/ewdapps/smartVistADemo> directory.

- if you are using Cache, use Studio to create a routine named *^smartVistADemo* in the namespace in which you want to run the application and copy the contents of the *smartVistADemo.m* file into it, then save and compile the routine
- if you are using GT.M, copy the *smartVistADemo.m* routine file to the directory you use for your own application routines (eg on a dEWDrop system, you'll find this routine file already pre-exists in the directory */home/vista/p/*)

The chapter after next will use this application as an example of how to build an EWD SMART Container in which to run SMART Apps.

Maintaining SMART Manifests

About SMART Manifests

A key part of a SMART App is its manifest. A SMART manifest is a JSON object that contains meta-data describing the SMART App, in particular:

- **id:** its unique Id
- **index:** the URL from which it can be fetched

Details on SMART App Manifests can be found at http://dev.smartplatforms.org/reference/app_manifest/

For example, Harvard Medical School's *GotStatins* application (which they describe as the Hello World of SMART Apps) is as follows:

```
{
  "author": "Josh Mandel, Children's Hospital Boston",
  "description": "Determines whether patient is taking a statin",
  "icon": "http://sample-apps.smartplatforms.org/framework/got_statins/icon.png",
  "id": "got-statins@apps.smartplatforms.org",
  "index": "http://sample-apps.smartplatforms.org/framework/got_statins/index.html",
  "mode": "ui",
  "name": "Got Statins?",
  "requires": {
    "http://smartplatforms.org/terms#Medication": {
      "methods": ["GET"]
    }
  },
  "scope": "record",
  "version": ".1a"
}
```

SMART App authors must create and make available a manifest for each SMART App. In order to run a SMART App you must obtain a copy of its manifest, either by contacting its author or a web-site that contains the manifest(s) you require.

Note: on dEWDrop systems, a number of manifests have been pre-installed.

EWD's SMART Container is designed to be able to invoke automatically any SMART App for which a manifest has been registered. All you need to do is obtain and save a manifest: this provides enough information to EWD in order to run it within its SMART Container.

Registering Manifests on an EWD System

The SMART EWD application includes a sub-system for registering and maintaining SMART Manifests. The manifest maintenance application is an ExtJS v4 application, so you must have installed and compiled EWD and its SMART Container components as described in the previous chapter. You can run the manifest maintenance application from a browser by invoking a URL similar to:

- <http://192.168.1.100/vista/SMART/manifests.ewd>

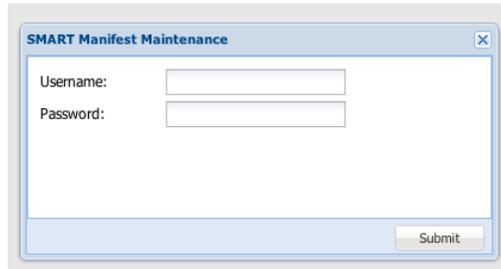
The exact URL you use will depend on:

- the IP address or domain name of the web-server that front-ends your EWD applications. The above example assumes it is 192.168.1.100
- the EWD *RootURL* configuration. On a dEWDrop system this is */vista/*, but it is often configured as */ewd/*

Otherwise the latter part (*/SMART/manifests.ewd*) will normally be the same.

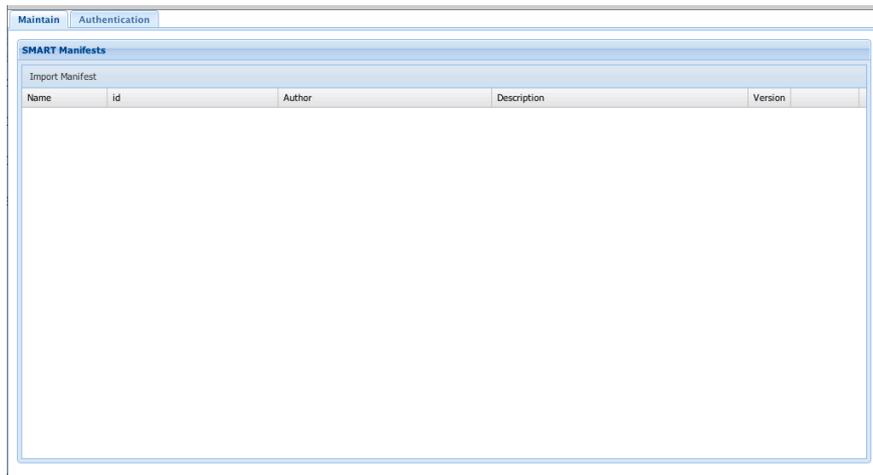
Note: on a dEWDrop system you must use SSL, so the URL should commence with *https://* rather than *http://*

If everything is working correctly, you should see a login screen:

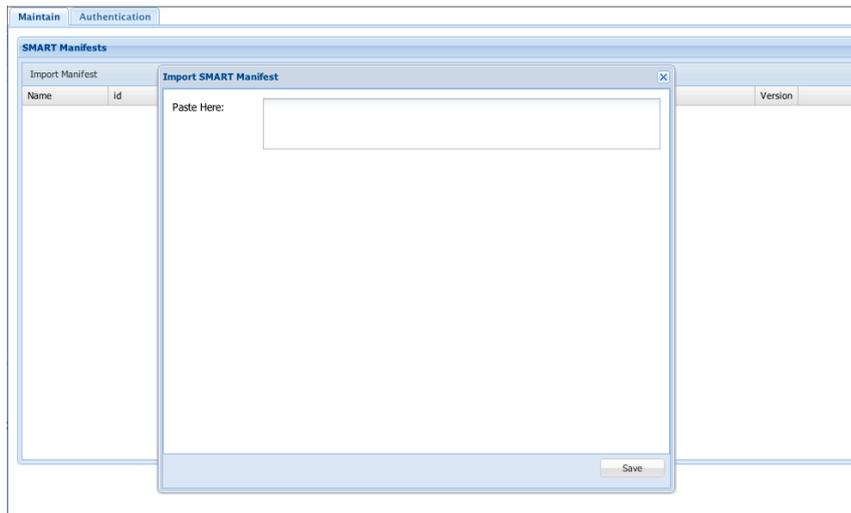


Note: if you first get a warning page telling you that the site's security certificate is not trusted, click *Proceed anyway*.

The initial default username/password is *admin/admin*. Click the Submit button and you should see an empty grid panel:



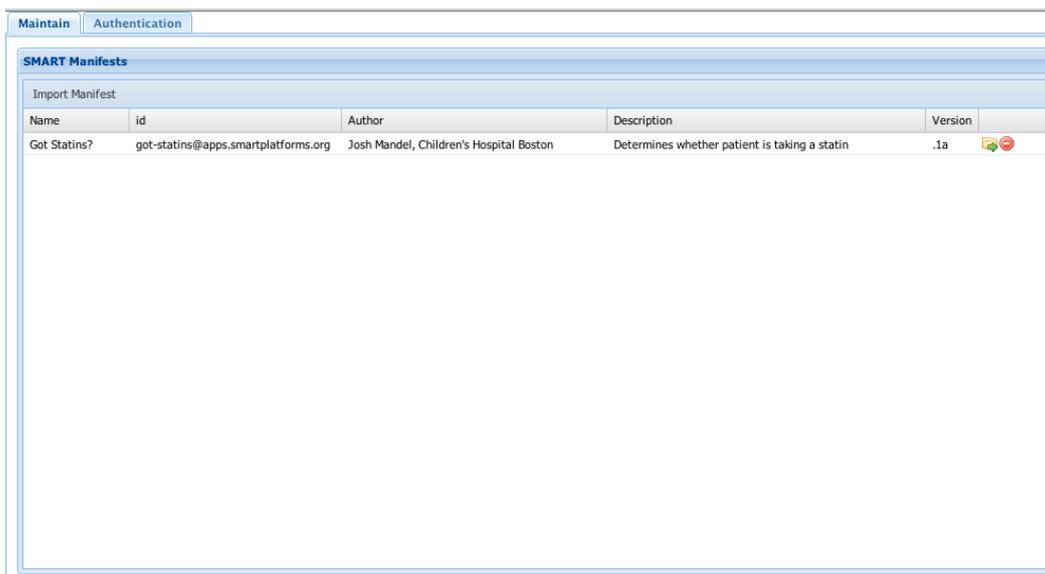
Locate the Import Manifest button (just under the grid's title) and click it: a pop-up window should appear:



Cut and paste a manifest's contents into the field where it says *Paste Here*, eg try the *Got Statins?* manifest:

```
{
  "author": "Josh Mandel, Children's Hospital Boston",
  "description": "Determines whether patient is taking a statin",
  "icon": "http://sample-apps.smartplatforms.org/framework/got_statins/icon.png",
  "id": "got-statins@apps.smartplatforms.org",
  "index": "http://sample-apps.smartplatforms.org/framework/got_statins/index.html",
  "mode": "ui",
  "name": "Got Statins?",
  "requires": {
    "http://smartplatforms.org/terms#Medication": {
      "methods": ["GET"]
    }
  },
  "scope": "record",
  "version": ".1a"
}
```

Click the Save button and you should see the manifest registered in the grid:



You can use the two icons at the right hand side to view and delete the manifest from the EWD system.

Repeat the process to add as many manifests as you like.

You are now ready to create a SMART Container in which to run the SMART Apps for which you've registered their manifests.

Creating a SMART Container

The Basics of a VistA SMART Container

A typical SMART Container-based application for a VistA system will consist of:

- a login form, in which the user must identify themselves to the VistA system using an Access Code & Verify Code
- once the user is logged in, a means of identifying an individual patient that is registered in the VistA system. SMART Apps currently display information for individual patients only
- a main work page that must include a work area in which an *iframe* can be created by EWD. The SMART App(s) will run in this *iframe*.
- a means of identifying the manifest for the SMART App you want to run against the selected patient. This should be done either implicitly within the application's logic, or explicitly by the user selecting the manifest in some way, eg from a list or by clicking/tapping a button.

A good example is the *smartVistADemo* application that was described in a previous chapter. In this chapter we'll examine this application in detail. By the end of this chapter you should be able to understand all the procedures and steps you need to follow in order to construct any SMART Container-based application.

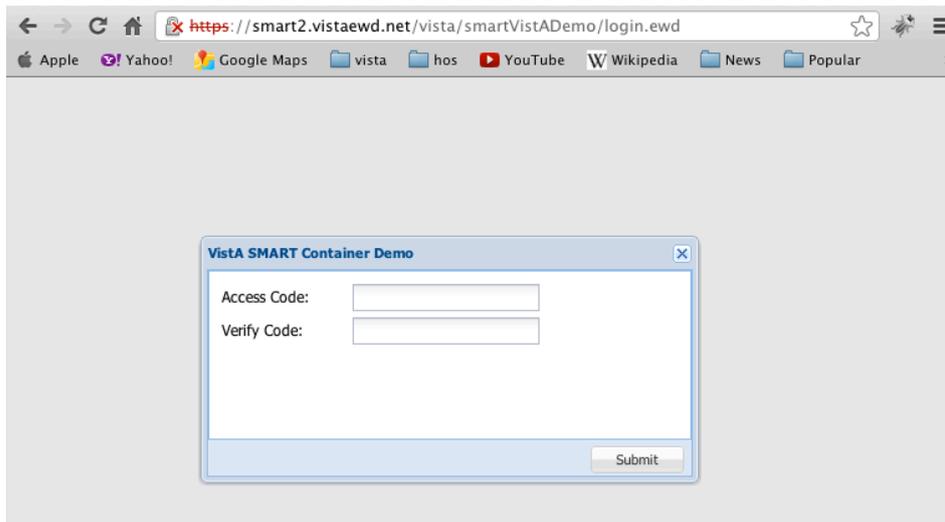
Note that the *smartVistADemo* application is an ExtJS v4-based application, so you must ensure that you have installed the latest version of the ExtJS Javascript libraries on your system.

The Login Form

The first page of our application is its Login Form. Here is the EWD Container page (*login.ewd*) that contains it:

```
<ext4:container rootPath="/vista/ext-4" title="SMART Container Demo">
<ext4:modalwindow title="VistA SMART Container Demo" height="200" width="400" layout="fit" autoShow="true">
<ext4:formPanel bodyPadding="10">
  <ext4:textfield id="accessCode" inputType="password" fieldLabel="Access Code" allowBlank="false" value="" />
  <ext4:textfield id="verifyCode" inputType="password" fieldLabel="Verify Code" allowBlank="false" value="" />
  <ext4:submitbutton text="Submit" nextPage="loginRedir" />
</ext4:formPanel>
</ext4:modalwindow>
</ext4:container>
```

This is a very simple EWD/ExtJS page which, when run, will bring up the Access Code/Verify Code prompts in a modal pop-up window:



As you'll see from the `<ext4:submitButton>` tag, the EWD fragment that handles this login-form (determined by the attribute `nextPage`) is `loginRedir.ewd`. This fragment does two things:

- its `onBeforeRender` method validates the Access Code & Verify Code that the user entered and either returns an error or signals that the user's credentials are valid
- it redirects to a second EWD Container page (specified as not being a first page and therefore inaccessible except via an EWD-generated, tokenised URL). The page to which it redirects is the main SMART Container page.

Here's `loginRedir.ewd`:

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

Its `onBeforeRender` method invokes the standard VistA code for confirming an Access Code / Verify Code:

```

login(sessid)
;
n DT,DUZ,IO,IOF,IOM,ION,IOS,IOSL,IOST,IOT,PNAME,POP,U,USRNAME
n V4WACC,V4WAVC,V4WVAVC,V4WTR,V4WUSER,V4WVCC,V4WVMSG,V4WVER,X
;
s U="^^"
d NOW^%DTC s DT=X
s (IO,IO(0),IOF,IOM,ION,IOS,IOSL,IOST,IOT)="",POP=0
;
s V4WACC=$$getRequestValue^%zewdAPI("accessCode",sessid)
i V4WACC="" QUIT "Please enter your Access Code"
;
s V4WVER=$$getRequestValue^%zewdAPI("verifyCode",sessid)
i V4WVER="" QUIT "Please enter your Verify Code"
s V4WAVC=V4WACC_"_";_V4WVER,V4WAVC=$$ENCRYP^XUSRB1(V4WAVC)
;
d SETUP^XUSRB()
d VALIDAV^XUSRB(.V4WUSER,V4WAVC)
s V4WVCC=V4WUSER(0)
;
s V4WVCC=$g(V4WUSER(2))
s V4WVMSG=$g(V4WUSER(3)) ;sign in message
;
s V4WTR=""
i 'V4WVMSG,$g(DUZ) s V4WTR=": " $$GET1^DIQ(200,DUZ_",",9.4) ;Termination reason
i 'V4WVMSG QUIT V4WUSER(3)_V4WTR ; invalid login
;
; Access Code / Verify Code is OK
;
s USRNAME=$p(^VA(200,V4WVMSG,0),"^")
s PNAME=$p(USRNAME,"",2)_"_"_$p(USRNAME,"")
;
d setSessionValue^%zewdAPI("DT",DT,sessid)
d setSessionValue^%zewdAPI("DUZ",V4WVMSG,sessid)
d setSessionValue^%zewdAPI("vista_user.name",USRNAME,sessid)
d setSessionValue^%zewdAPI("vista_user.id",V4WVMSG,sessid)
d setSessionValue^%zewdAPI("displayName",PNAME,sessid)
QUIT ""

```

The key part to note is at the very end of this function (shown in bold): if the user has provided a valid Access Code / Verify Code, a number of key Vista variables are saved into the EWD Session. These are required for later transactions within the application. The *vista_user* prefix is specific to the EWD SMART Container's Session variables.

The Main Layout Page

If the user enters a valid Access Code/Verify Code, they are redirected to *index.ewd* which is as follows:

```

<ext4:container rootPath="/vista/ext-4" isFirstPage="false" smartContainer="true">
  <ext4:viewPort layout="border">

    <ext4:panel region="north" id="northPanel" title="SMART Vista Demo" autoheight="true" border="false"
      margins="0 0 5 0">
      <ext4:toolbar dock="top">
        <ext4:textItem text="Patient: " />
        <ext4:textItem text="Not Selected" id="patientId" />
        <ext4:fill />
        <ext4:button text="Select Patient" nextPage="selPatient" />
      </ext4:toolbar>
    </ext4:panel>

    <ext4:panel region="west" collapsible="true" title="Manifests" width="200" autoheight="true"
      addPage="manifestTree" />

    <ext4:panel region="center" id="smartPanel" />

  </ext4:viewPort>
</ext4:container>

```

This page creates the overall layout for the application. Note the special attribute that we use in the `<ext4:container>` tag: `smartContainer="true"`. This tells EWD's compiler to add in the extra logic required for the SMART Container.

The second feature to note is the toolbar button that allows us to select a patient from the VistA system. When this is clicked, the fragment *selPatient.ewd* is invoked.

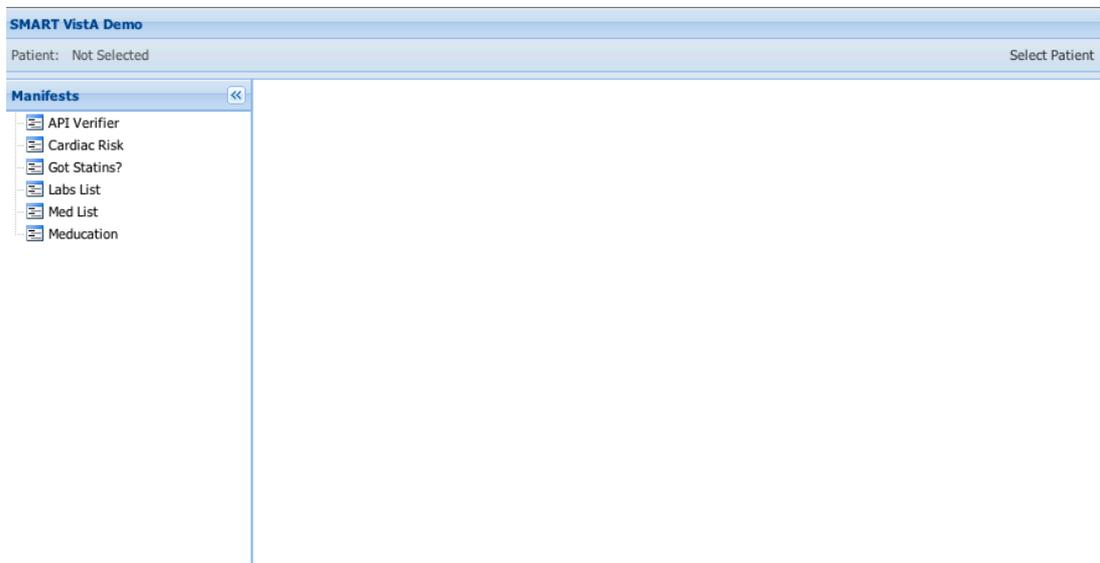
Finally, in this demo application, the west panel is populated by a fragment named *manifestTree.ewd*. This fragment is as follows:

```
<ext4:fragment onBeforeRender="getManifests^smartVistADemo">
  <ext4:treepanel border="0" id="manifests" autoheight="true" width="205" sessionName="SMART_manifests"
    nextPage="runSmartApp" addTo="smartPanel" replacePreviousPage="true" expandedTree="true" />
</ext4:fragment>
```

The purpose of this fragment is to create a tree menu, showing all the manifests that are registered on the VistA system. The EWD Session Array that populates the tree is named "SMART_manifests" and is created by the fragment's *onBeforeRender* method which is as follows:

```
getManifests (sessid)
  n id,manifest,no
  ;
  s id="",no=0
  f s id=$o(^SMART("manifest",id) q:id="" d
  . s no=no+1
  . s manifest(no,"text")=^SMART("manifest",id,"name")
  . s manifest(no,"nvp")="manifestId="_id
  d mergeArrayToSession^%zewdAPI(.manifest,"SMART_manifests",sessid)
  QUIT ""
  ;
```

So, when the user successfully logs into the application, the following screen should appear (the list of manifests in the tree menu will depend on which ones you've registered):



Patient Selection

Before running any of the SMART Apps, the user must select a patient by clicking on the toolbar button. *selPatient.ewd* is the fragment that handles this event:

```

<ext4:fragment onBeforeRender="initialisePatientList^smartVistADemo">
  <ext4:window title="Select Patient" id="selectPatientWindow" autoShow="true" height="300" width="300">
    <ext4:formPanel bodyPadding="10" height="150">
      <ext4:comboBox id="patCombo" fieldLabel="Patient Name" enableKeyEvents="true" name="patientName" value="">
        <ext4:listeners>
          <ext4:listener afterrender="function() {this.triggerWrap.dom.style.display = 'none';}" />
          <ext4:listener keyup="function(txt) {var value = txt.getValue(); this.triggerWrap.dom.style.display =
''; var nvp='comboId=patCombo&text=' + value; EWD.ajax.getPage({page:'updateCombo',nvp:nvp});}" />
          <ext4:listener blur="function() {if (this.getValue() === null) this.triggerWrap.dom.style.display =
'none';}" />
        </ext4:listeners>
      </ext4:comboBox>
      <ext4:submitButton text="Select" nextPage="setPatient" />
    </ext4:formPanel>
  </ext4:window>
</ext4:fragment>

```

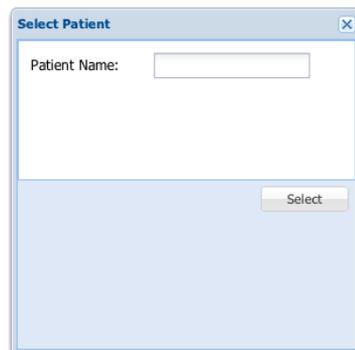
Its onBeforeRender method is as follows:

```

initialisePatientList(sessid)
d clearList^%zewdAPI("patientName",sessid)
d appendToList^%zewdAPI("patientName","Select Name","null",sessid)
QUIT ""
;

```

This fragment brings up a combo box inside a pop-up window into which the user begins typing the required patient name.



On every keystroke, the fragment *updateCombo.ewd* is invoked to update the combobox list with matching names based on the characters typed so far. This fragment is as follows:

```

<ext4:fragment onBeforeRender="updateCombo^smartVistADemo">
  <ext4:js at="top">
    EWD.ext4.updateCombo = function(comboId,values) {
      var comboStore = Ext.getCmp(comboId).store;
      var value;
      comboStore.loadData(values, false);
    };
    var comboId='<?=#comboId ?>';
    var values = <?=#values ?>;
    EWD.ext4.updateCombo (comboId,values);
  </ext4:js>
</ext4:fragment>

```

Its *onBeforeRender* method returns a list of matching patient names formatted as a Javascript array and held in the EWD Session variable named *values*:

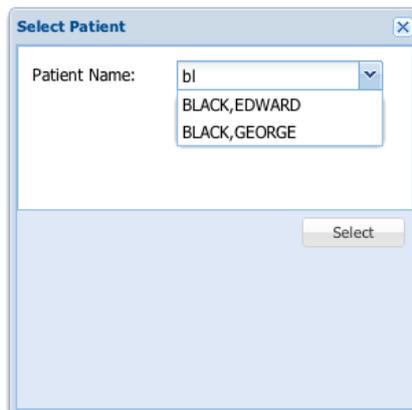
```

updateCombo (sessid)
n comma,no,options,prefix
;
s prefix=$$getRequestValue^%zewdAPI ("text",sessid)
s comboId=$$getRequestValue^%zewdAPI ("comboId",sessid)
d setSessionValue^%zewdAPI ("comboId",comboId,sessid)
d deleteFromSession^%zewdAPI ("options",sessid)
d comboList (prefix,.options)
d mergeArrayToSession^%zewdAPI (.options,"options",sessid)
s values="["
s no="",comma=""
f s no=$o(options(no)) q:no="" d
. s values=values_comma_"{name:'"_options(no,"displayText")_'",value:'"_options(no,"value")_'}"
. s comma=", "
s values=values_""]"
d setSessionValue^%zewdAPI ("values",values,sessid)
QUIT ""
;
comboList (prefix,options) ;Generates the list of patients for the combo box
n id,no,name,stop,U
;
s U=""^"
s prefix=$$zcvt^%zewdAPI (prefix,"U")
;
;Generate the list of patients in the combo box to match what the user typed
;Using the patient XREF
;
k options
s name=prefix
i prefix="" s name=$O(^DPT("B",prefix),-1)
s no=1,stop=0
f s name=$o(^DPT("B",name)) quit:name="" d q:stop
. i $e(name,1,$l(prefix))'=prefix s stop=1 q
. s id=""
. f s id=$o(^DPT("B",name,id)) q:id="" d q:stop
. . s no=no+1 i no>500 s stop=1 q
. . s options (no,"displayText")=name
. . s options (no,"value")=no
. . s options (no,"id")=id
QUIT
;

```

Note the way the formatted name is cached in the options array and retained in the Session.

With just a few key-strokes, this combobox allows us to identify the patient required:



Then just click on the matching name and click the *Select* button. This invokes the fragment *setPatient.ewd* which is as follows:

```
<ext4:fragment onBeforeRender="setPatient^smartVistADemo">
  <ext4:js at="top">
    Ext.getCmp('patientId').setText('<?=#vista_patient.name ?>');
    Ext.getCmp("selectPatientWindow").destroy();
    Ext.getCmp("smartPanel").removeAll(true);
  </ext4:js>
</ext4:fragment>
```

This fragment updates the *Patient Name* field in the main layout's toolbar, by using the value in the EWD Session Variable *vista_patient.name*. This is created by the fragment's *onBeforeRender* method:

```
setPatient(sessid)
;
n id,name,no,options
;
s no=$$getSessionValue^%zewdAPI("patientName",sessid)
d mergeArrayFromSession^%zewdAPI(.options,"options",sessid)
s id=$g(options(no,"id"))
s name=$g(options(no,"displayText"))
d setSessionValue^%zewdAPI("vista_patient.id",id,sessid)
d setSessionValue^%zewdAPI("vista_patient.name",name,sessid)
QUIT ""
;
```

Note the way we pick up the formatted name for the selected patient from the cached copy in the options Session Array.

Finally, the combobox pop-up window is removed by the Javascript in the *setPatient.ewd* fragment and anything in the center panel is deleted. Now the main window is ready for us to select a SMART manifest (in the example below for a patient named Edward Black):



Running a SMART App via its Manifest

Everything that now happens is controlled from the tree menu. This, as described earlier, was generated by the fragment *manifestTree.ewd*:

```
<ext4:fragment onBeforeRender="getManifests^smartVistADemo">
  <ext4:treepanel border="0" id="manifests" autoheight="true" width="205" sessionName="SMART_manifests"
    nextPage="runSmartApp" addTo="smartPanel" replacePreviousPage="true" expandedTree="true" />
</ext4:fragment>
```

When a manifest name is clicked, the fragment named *runSmartApp.ewd* is invoked (*nextPage="runSmartApp"*), and its output is directed into the panel whose id is *smartPanel* (*addTo="smartPanel"*). If you look back at the main layout page (*index.ewd*), you'll see that this is the center panel.

So let's take a look at *runSmartApp.ewd*. This is the crucial page in our example and should be used as a template for SMART Container-based applications:

```
<ext4:fragment onBeforeRender="getManifest^smartVistADemo">
  <ext4:js at="top">
    function loadSmartApp(divId) {
      SMART.simple_context = {
        record: {
          full_name: '<?=#vista_patient.name ?>',
          id: '<?=#vista_patient.id ?>'
        },
        user: {
          full_name: '<?=#vista_user.name ?>',
          id: '<?=#vista_user.id ?>'
        },
      };
      SMART.launchApp('<?=#SMART_manifest.id ?>',divId);
    }
  </ext4:js>
  <ext4:panel border="0" smartFrameDiv="smartFrame" height="100%" width="100%"
    title="<?=#SMART_manifest.name ?>">
    <ext4:listeners>
      <ext4:listener afterrender="loadSmartApp('smartFrame')"/>
    </ext4:listeners>
  </ext4:panel>
</ext4:fragment>
```

We need to also look at its *onBeforeRender* method:

```
getManifest(sessid)
;
n id
;
s id=$$getRequestValue^%zewdAPI("manifestId",sessid)
i id="" QUIT "No Manifest selected!"
i '$d(^SMART("manifest",id)) QUIT "Unknown Manifest"
s patientId=$$getSessionValue^%zewdAPI("vista_patient.id",sessid)
i patientId="" QUIT "You must first select a patient"
d setSessionValue^%zewdAPI("SMART_manifest.id",id,sessid)
s name=$g(^SMART("manifest",id,"name"))
d setSessionValue^%zewdAPI("SMART_manifest.name",name,sessid)
QUIT ""
;
```

This works as follows: when a manifest name is clicked in the tree menu, it is identified in the *^SMART* global in which it is registered via its id. This id is sent in the incoming request as the name/value pair named *manifestId*. The name and id of the manifest are then saved in the EWD Session as *SMART_manifest.name* and *SMART_manifest.id* respectively.

The *runSmartApp.ewd* page then proceeds as follows:

- an ExtJS panel is created and inserted into the layout's center panel. Note the special attribute *smartFrameDiv* that tells EWD to create an iframe. The value of the *smartFrameDiv* attribute specifies the *id* of the iframe that will be created
- When the panel is rendered, its *afterrender* event fires which invokes the *loadSmartApp* Javascript function. Note that we must pass the same frame id as defined in the panel's *smartFrameDiv* attribute (in this example: *smartFrame*)
- The Javascript function *loadSmartApp* instantiates the *SMART.simple_context* object and then launches the SMART App via the manifest id held in the EWD Session Variable *SMART_manifest.id*.

That's it: at this point the standard SMART App loader takes over: the SMART App is launched and runs inside the iframe that EWD has automatically generated inside the newly created panel.

Note the way the *SMART.simple_context object* is defined using EWD Session variables holding:

- the user name and id, established when the Access Code / Verify Code was entered by the user
- the patient name and id, established when the patient name was selected from the combobox

ie:

```
SMART.simple_context = {
  record: {
    full_name: '<?=#vista_patient.name ?>',
    id: '<?=#vista_patient.id ?>'
  },
  user: {
    full_name: '<?=#vista_user.name ?>',
    id: '<?=#vista_user.id ?>'
  },
};
```

So, if we clicked the Got Statins manifest name in the tree, we should see the following:



You now have a working SMART Container for your VistA system.

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 dEWDDrop Virtual Machine (<http://www.fourthwatchsoftware.com>) which is pre-configured with the latest build of EWD, comes pre-configured with a working VistA system and SMART Container and includes a set of working SMART Applications.

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 do 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:

Edit definition for CSP application /csp/ewd:

[General] Application Roles Matching Roles

CSP Application Name*: /csp/ewd (e.g. /csp/appname)

Description: EWD Applications

Enabled:

Resource required to run the application:

Allowed Authentication Methods: Unauthenticated
 Password

Accept sessions established by other CSP applications:

Namespace: USER

CSP Files Physical Path: c:\intersystems\cache\csp\ewd\

Recurse: Yes

Auto Compile: Yes

Event Class:

Default Timeout: 3600

Default Superclass:

Use Cookie for Session: Autodetect

Session Cookie Path: /csp/ewd/

Serve Files: Always Serve Files Timeout: 3600

Lock CSP Name: Yes

Custom Error Page: /csp/samples/error.csp

Package Name:

Login Page:

Change Password Page:

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 (MGWLPM) 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^0%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")
```

GTM:

```
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")
```

GTM:

```
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](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=\[pageName\]](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](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>