# Notes for Development:

# How Tos:

## Overall Functionality

There are three main javascript files that deal with generic medCafe functionality:

* **medCafe.js**
* **medCafeTabs.js**
* **medCafe.widgets.js**

Between these the majority of non widget specific medCafe functionality is dealt with. i.e. deals with adding of a new tab, drag and dropping of tabs, saving of tabs (very complex.).

Most other widgets have their own javascript file following this convention:

medCafe.<widgetname>.js

e.g. medCafe.allergies.js

## How Tabs get Loaded

In index.jsp, there is a call to loadWidgetData for both patient specific and general widgets

This calls **loadWidgetData** in **medCafe.widget.js**

**loadWidgetData** makes a call to **widgets-listJSON.jsp** which lists all the widgets based on the **WidgetList.xml** document (located under config.)

Brings back the following JSON as an example:

{“widgets”:[

…

{"id":1,"repository":"OurVista","name":"Timeline","image":"images/timeline.png","method":"","clickUrl":"timelineJSON.jsp","params":"","type":"Timeline"}

…]

}

This is then turned into html using the template,

listWidgets.vm (and associated v2js\_listWidgets javascript).

Resulting in html of the following format:

<div class=”imageContain”>

<img custom:repository="OurVista" custom:params="" custom:id="1" custom:type="Timeline" custom:url="timelineJSON.jsp" alt="Timeline" src="images/timeline.png">

</div>

These are the major parameters used for further processing.

* **Repository** : tells the widget wether this is local, OpenVista, or hData.
* **Type**: the type of widget, very important parameter for further processing.
* **Url**: where to get associated data for this widget (usually , but not always JSON data).
* **Src**: The image to display for the widget.

Then for each “imageContain” class a ‘startWidgetDrag” function is associated with the mouseDown event (touchmove for case of iPad).

This calls ‘startWidgetDrag’ in medCafe.js.

This clones the image object and allows fr dragging to anywhere on screen. (Mainly holdover from when the widget listing was in its own iFrame.)

This then allows for dragging of the cloned image object, until a drop event is detected on the ‘droppable area’. This is an area inside the tabs\_template.jsp object.

A “widgetContent” class.

(This is why we always need at least an empty tab- so as to be able to register a ‘dropped’ event of a widget).

Inside of tabs\_template.jsp is a method that triggers on dropping of an image object (maybe need to check that this is of type imageContain class?)

This method beginning:

$(".widget-content").droppable({

drop: function(event, ui)

{…

Checks if this is a valid object to drop, i.e. is an imageContain object, with associated meta data embedded in the html (see above).

The code then parses out the html data embedded to extract all the meta data needed for further processing. E.g. widget type and url, etc,.

It then looks, using the repository parameter for the associated patient id, from that repository. Every repository will probably have its own id for patients.

Then a check is carried out to see if this tab is empty or contains a widget.

If it does contain a widget, then a new tab will be created.

If it doesn’t then the current tab will be used for the new widget.

The most common case is that widget has content:

This calls the function:

createLink(patientId,link, text, type ,params, repository, repPatientId);

CreateLink in medCafeTab.js is the major method for creation of any new Widget.

medCafeTabs.js CreateLink function :

Creates a new Tab, calling addTab method.

The addTab method, uses the label and the type of widget to create a new empty tab object. It then loads a new **tabs-template.jsp** object into this tab. And then selects the new tab to have focus.

After the new tab object has been added, and loaded with the tabs-template.jsp data, the next step is to create the widget specific data.

This is done through calling createWidgetContent function on medCafe.js

medCafe.js createWidgetContent

This method determines what functions and code to call for the widget, based upon the widget type.

Current types are:

* Chart\*
* Image\*
* Detail
* Viewer\*
* Annotate\*
* Repository
* Bookmarks
* Medications
* History
* Problem
* Immunizations
* Allergies
* Filter\*
* Timeline\*
* Symptoms\*
* AddHistory\*
* EditorNonIFrame\*
* Support
* SingleImage

Of these the ones marked with (\*)call the medCafeTabs.js **addWidgetTab** function. The goal is to have all widgets go through this function. Removing necessity of having different code for different widgets. (Currently 9 out of the 19 widgets call this method.)

The method **populateWidgetSettings** is then called.

This calls populateExtWidgetSettings in medCafe.widgets.js.

This loads the widget settings into an array, for use in saving to the database. And contains such information such as type, tab\_number, url, repository, label, patient\_id and repository\_patient\_id.

(The retrieval happened through loadExtWidgetSettings, and is called on changing the selected patient.)

## medCafeTabs.js addWidgetTab

Each tab has a ‘widget’ class with an id of "yellow-widget<%=tabNum%>" (look at tabs-template.jsp)

<div class="widget color-<%=tabNum%>" id="yellow-widget<%=tabNum%>">

(this is a holdover from initial development – the yellow –widget means nothing and can be changed, but is just a low priority).

Inside of this class is a ‘widget-content’ class.

<div class="widget-content no-copy" id="widget-content<%=tabNum%>">

<p><div id="aaa<%=tabNum%>" class="no-copy"></div></p>

<div id="dialog<%=tabNum%>">

<div id="modalaaa<%=tabNum%>"></div>

</div>

<div id="hasContent" custom:hasContent="false"></div>

</div>

Of particular note is that this has an id of “**aaa<tabnumber>”** e.g. “aaa2” for tab number 2 etc,. This is key to all functionality, as the aaa number is heavily used throughout the code.

The addWidgetTab function, first builds a string to access the data specified in the server link, this is the url specified in the WidgetList.xml file.

var serverLink = server + "?repository=" + repId + "&patient\_id=" + patientId + "&patient\_rep\_id=" + patientRepId;

$.get(serverLink, function(data)

{

And then does an Ajax get. Bringing back the url (a jsp) , and then appending it to the widget-content using the **aaa** number.

Then any widget specific data is handled through the **processScripts** method.

This method checks to see if the associated widget specific has been loaded, e.g. medCafe.timeline.js, and loads it dynamically if this hasn’t already been added.

Then it calls a function of format

Process<Type>(repId, patientId, patientRepId, data, type,tab\_num)

e.g.

processTimeline(repId, patientId, patientRepId, data,type, tab\_num);

This will processing script, will call any methods that should be called after the associated jsp file has been loaded (I.e. the HTML DOM object has finished loading).

It is within this processing script that all widget specific functionality should be called.

The last step in the addWidgetTab is to add a wrapper to allow for scrolling of the content of the widget.

## To Add a New Widget using addWidgetTab

1. Create a jsp object that contains the desired layout of the widget e.g. look at the list in the WidgetList.xml for list of all the jsps that are currently used.

e.g. if new widget is called “grommit” then jsp might be listGrommits.jsp

Note that due to Safari’s loading javascript slightly differently from Firefox, in the case of jsps, no javascript should be contained in the jsp itself. All javascript functionality should be contained in the medCafe.<widgetName>.js process<widgetName> method. (As Safari will not run the javascript contained within a dynamically loaded jsp.)

2. Create a file **medCafe.<widgetName>.js**, e.g. medCafe.grommit.js. Which should have, at minimum a **processGrommit**(….) method.

This should contain any processing to be carried out in javascript once the html DOM is fully loaded. (In some cases – notably Images, a delay will be needed to allow for the html to be fully loaded.)

3. Find/ Create a new Icon for the widget. Name according to <widget>.jpg or <widget>.png. e.g.

Grommit.png gromit.jpg

4. Make an entry in the WidgetList.xml file pointing to the jsp and image files for this widget type. Under either patientSpecific or general widget types.

For example:

<medCafeWidget>

<name>Grommit</name>

<type>Grommit</type>

<image>grommit.png</image>

<server></server>

<clickUrl>listGrommits.jsp</clickUrl>

<method></method>

<repository>local</repository>

</medCafeWidget>

4. Make an appropriate entry in medCafe.js createWidgetContent method

(This if/else statement can be steamlined a bit).

For example:

else if (type == "Grommit")

{

if (typeof processGrommit == 'undefined')

{

$.getScript('js/medCafe.grommits.js', function()

{

addWidgetTab(this, link, tab\_num, patientId, repId, patientRepId, type);

});

}

else

{

addWidgetTab(this, link, tab\_num, patientId, repId, patientRepId, type);

}

}

5. Make an entry in medCafeTabs.js processScripts method

else if ( type == "Chart")

{

processGrommit(repId, patientId, patientRepId, data, type,tab\_num);

}

Note: The check for the script existence can take place in either medCafe.js createWidgetContent or in the medCafeTabs. processScripts method (it doesn’t really matter.)

## Adding the JSON data:

1. Create java object under restlet package with the following name convention:

<WidgetName>Resource.java

e.g. GrommitListResource.java

2. Implement the toJson method to return the data in the expected JSON format:

@Get("json")

**public** JsonRepresentation toJson(){

**try**

{

3. In the MedCafeApplication class add an entry in the createInboundRoot method (this could maybe later be populated by a config file)

router.attach("/repositories/{repository}/patients/{id}/grommit", org.mitre.medcafe.restlet.ListGrommitResource.**class**);

4.Create a jsp that returns the JSON data, and would contain something similar to code listed below.

e.g. grommitListJSON.jsp

<%@ page import="org.mitre.medcafe.util.\*" %>

<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>

<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>

<%

String patient\_id = request.getParameter("patient\_id");

if (patient\_id == null)

patient\_id = "1";

String repository = request.getParameter("repository");

if (repository == null)

repository = "OurVista";

String jspUrl = "/repositories/" + repository + "/patients/" + patient\_id + "/grommit";

String user = request.getRemoteUser();

jspUrl = jspUrl + "?user=" + user;

%>

<tags:IncludeRestlet relurl="<%=jspUrl%>" mediatype="json"/>

5. Call the JSON data from within process<widget> method using a similar method as listed below

var serverLink = "grommitListJSON.jsp?patient\_id=" + patientId + "&image=" +server;

$.getJSON(serverLink, function(data)

{

//Check to see if any error message

if (data.announce){

updateAnnouncements(data);

return;}

//If no error message then continue with javascript

var html = v2js\_listGrommits( data );

$("#grommit").html(html);

……

});

6. Create the associated listGrommit.vm file, and call appropriately in the processGrommit method. To produce the desired format for the html data.

Now you should have a fully functioning new Grommit widget. That you can drag and drop from the Widget list on the right hand side of the medCafe application.

## Overview of how MedCafe Retrieves Tab information on Patient Select:

In **index.jsp** the **searchPatient.jsp** is loaded via iFrame.

This calls **setOnSelect** method in **medCafe.patients.js**, which checks to see if this is called from introPage or not. (The searchPatient.jsp) is used in both places.

If it is in the index page, then calls **retrieve** on medCafe.patients.js.

### Retrieve for patient on medCafe.patient.js

This method gives the user a dialog option, if they want to save, not save and continue to new patient, or cancel the change of patient.

If they decide to save, calls **saveWidgets** on medCafe.widget.js (see below).

Once the data for the previous patient is saved, the tabs are removed, by calling closeAllTabs.

The **populate** method on **medCafe.patients.js** is now called for the newly selected patient.

The populate method calls the **retrievePatient.jsp** via a Ajax getJSON (jQuery) call.

The retrievePatient.jsp gets a list of widgets for this patient from the database table **widget\_params**, formatted as a JSON object.

It also adds the newly retrieved patient to the list of recently accessed patients (to be used for quick retrieval) and saves to recent\_patients table.

Then, the list of patient ids in each existing repository is retrieved and saved to a JSON object, “repPatientsIds” that is stored in session memory.

The JSON object containing the widget settings is then returned, example below:

{"widgets":

[{"id":2,"rep\_patient\_id":"4","location":"center","repository":"local","name":"Filter","server":"filterTools.jsp","patient\_id":4,"type":"Filter","tab\_order":2},

…..]}

If no widgets currently exist in the database (for case of new patient, or one not previously accessed in medCafe), then an empty tab is returned. This empty tab object is required for drag / drop functionality (see earlier).

The JSON object containing the widget settings is parsed for all required meta data.

e.g. the widget type, tab number, url, location (in tab order), repository, etc,..

Then using **addTab** method on **medCafeTabs.js** (described earlier in How Tabs Get Loaded section), a new tab is added to medCafe, and the **createWidgetContent** on medCafe.js is called for the new widget. (Also described earlier ).

## Saving the state of the Tabs/ Widgets saveWidgets on medCafe.widget.js

First the existing widgets are deleted from the database, for this patient using **deleteWidget.jsp**. To clear the database. (This has to happen prior to any further processing.)

Then get an array of all the Ids of the widgets that are currently associated with this patient (one per tab.), by calling **getAllIds** on medCafe.widgets.js.

e.g. id= “yellow-widget2”

Using this Id, the function **saveWidget** in **medCafe.widgets.js** is called

The saveWidget function uses the id to get a list of all the settings associated with this widget, and formats them as key value pairs used as input for **saveWidget.jsp**.

The saveWidget.jsp, takes these key value pairs and formats them as **JSON** data, example below.

{"id":"2","remove":"false","location":"center","repository":"OurVista","rep\_patient\_id":"2","name":"Details","clickUrl":"http://127.0.0.1:8080","server":"repository-listJSON.jsp","patient\_id":"2","type":"Detail","tab\_order":"2"}

This contains all the information needed to retrieve the widget at a later point.

The saveWidget.jsp then calls the Widget.saveWidgets(userName, jsonobj) java method, using this JSON Object, which saves the widget settings to the database in **widget\_params** table.

## To Add a new Filter

How system currently used Filters:

### To Set up a Filter Listener method

When the tabs-template.jsp is loaded, it calls a method **filterType**. This method uses the type of the widget to dynamically load a script for calling of a method for filtering the data.

The format follows a standard:

The script should be called:

js/filterDate<type>.js

and the method should be called:

filterDate<%=type%>(startDate, endDate,filterCat, tabNum );

(The ‘Date’ part of the method name is really a hold over from earlier development, as can be seen, in the argument list that filtering can take place on a date, or a string category. Refactoring out this name should be straightforward but low priority.)

e.g. For the ‘grommit’ widget:

The script following script should be created **js/filterDateGrommit.js**, and the method should have the following declaration

**filterDateGrommit(startDate, endDate, filterCat, tabNum).**

Look at filterDateImage.js script for an example of how this functionality can be used. (Which calls medCafe.images.js, filterImages).

On tabs-template.jsp this method is bound to an Event, ‘FILTER\_DATE’.

The FILTER\_DATE listener is initialized on medCafe.js (line 41).

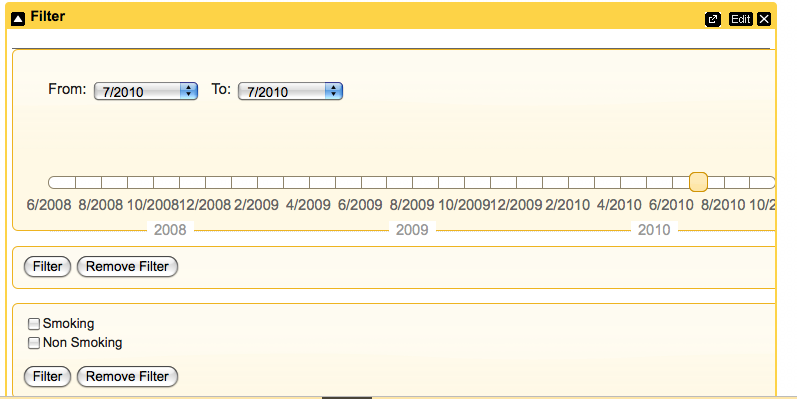
This allows for any widget that has been ‘registered’ for the FILTER\_DATE event to automatically call the associated method to filter the current data according to the new filter criteria.

### To Trigger a FILTER Event

Any widget can trigger this event. For example on clicking the ‘Filter’ button on the Filter widget.

In medCafe.filter.js, filterInitialize is an example of how to do this.

In the Filter widget, screen shot below:



Data can be filtered on date, or on a category.

On clicking the Filter button a method **triggerFilter** on **medCafe.js** will be called. This method will call the following single line.

$(document).trigger('FILTER\_DATE', [startDate, endDate, filterCat]);

Which, in turn will cause the FILTER\_DATE event to be called. Which will trigger the associated filter methods on any widgets that are set up for listening (registered) for this event

### Putting it together: How the Filter Criteria state is stored and used to filter Images in medCafe on trigger of the FILTER\_DATE Event.

In medCafe Filter widget whenever a Filter button is clicked, and the FILTER\_DATE event triggered, a call will be made to

1. setFilter.jsp
2. filterDateImage.js filterDateImage(startDate, endDate, tab\_num).

The setFilter.jsp will take the parameters, and build a MedCafeFilter object with the associated filter criteria.

This MedCafeFilter object is then stored into session memory under attribute name ‘filter’.

e.g. In setFilter.jsp:

session.setAttribute("filter", filter);

This MedCafeFilter object can then be accessed through any jsp, for example in **contentflow/ coverFeed.jsp** this MedCafeFilter object is accessed using the following line:

Object filterObj = session.getAttribute("filter");

The image JSON data, filtered by date and category is generated through calling the associated restlet on coverFeed.jsp

e.g. String url = "/repositories/medcafe/patients/" + patientId + "/images?start\_date=1/01/2009&end\_date=7/01/2010&user=gaily";

The image widget then uses this new JSON data to display only the required images using the method **filterDateImage** which is triggered on the FILTER\_DATE event. This will then generate the new set of images via the contentflow/coverFeed.jsp, by reinitializing the contentFlow object with the filtered JSON data from coverFeed.jsp using an Ajax get method call,

var fileUrl = "contentflow/coverFeed.jsp";

$.get(fileUrl, function(data)

{

See medCafe.images.js filterImages, which is called from filterDateImage.

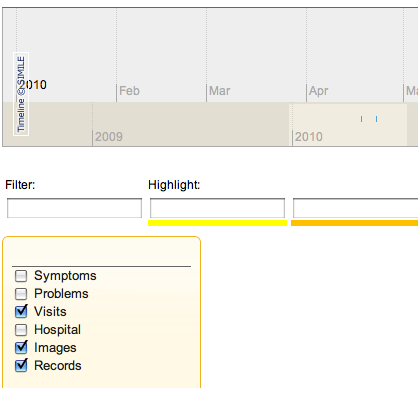
## To Add Events to the Timeline

The Events for the timeline are generated from the PatientListEventResource object under restlet package.

It calls a method:

Event.*retrieveEvents(…)*

Which takes the list of event types (‘Hospital’ ,’Records’,’Visits’,…) etc,.. as input. (Look at the list on the Timeline object.) and returns an ArrayList of Event objects.



The JSON is then generated for each Event and the result is returned in a JSON format that the timeline object can use.

To add the functionality to bring up a new Visit record:

Look at the following:

<http://simile.mit.edu/wiki/Timeline/Own_Handler_Instead_Of_Bubble>

Timeline.DurationEventPainter.prototype.\_showBubble = function(x, y, evt) { alert (evt.getDescription ()); }

Inside of medCafe.timeline.js

Is a method:

fillInfoBubbleCustom(evt, div, this.\_params.theme, this.\_band.getLabeller());

It is this method that should be used to generate the code for clicking on the link and bringing up the related data (whether image etc,) inside of medCafe.

The current example:

Timeline.OriginalEventPainter.prototype.\_showBubble = function(x, y, evt) {

var div = document.createElement("div");

var themeBubble = this.\_params.theme.event.bubble;

//Method to be used to create code to bring up the details inside of medCafe

fillInfoBubbleCustom(evt, div, this.\_params.theme, this.\_band.getLabeller(),patientId);

var link = evt.getLink();

//Rewrite the link - so that

SimileAjax.WindowManager.cancelPopups();

SimileAjax.Graphics.createBubbleForContentAndPoint(div, x, y,

themeBubble.width, null, themeBubble.maxHeight);

};

function fillInfoBubbleCustom(evt, elmt, theme, labeller, patientId)

{

var doc = elmt.ownerDocument;

var title = evt.getText();

var link = evt.getLink();

var image = evt.getImage();

if (image != null) {

//This is the code to display an Image in medCafe - may put this on the link click instead

displayImage(image, patientId, -1);

The displayImage method calls the code to add a new tab and display this image on the new tab.

Accessing the type of the widget, any functionality could be added here to display the associated information or widget on clicking the icon on the Timeline Widget.

# To Do List

Rework Repositories –

At the moment to get patient detail from say OpenVista, we have a widget PatientDetails, that only gets info from one repository, in this case OpenVista.

If you want patient details from hData then, another widget would have to be added to the right hand side widget listing. Obviously this is not desired functionality.

There are 3 options:

1. Have the user specify up front which repository he wants data from. The disadvantage of this approach is that the dctor may not know which repository the data he needs is in.

2. Have all data returned for this widget type from all repositories. Disadvantage is that now have to start handling error messages if results don’t return within certain time frame. Also have to notify the user that x repository is not available etc,..

May cause a performance issue.

3. Allow for each widget to return a specified set of data based on say a comma separated list in the repository parameter in ListWidgets.xml file. This would have the least impact on current functionality.

Annotate Images – zooming shapes, moving image (On Annotate):

To bring up this functionality, click on a Images, and click a document, then when the document is brought up in a new tab, click the Annotate button. (This is the only widget that is currently in iFrame.)

Tag user specified shapes with associated Notes. (Probably need to use a right click menu).

The image won’t drag when I have added the capability to draw on shapes on the image. Due to the fact that the mousedown event on the image has overwritten the mousedown on drag for the underlying canvas.

I have tried various methods, without success, so added navigation buttons at the bottom. Unfortunately the shapes do not always stay in sync with the underlying image.

Tag Clouds?

## Test with iPad

It’s been a while since I last tested with the iPad.

## Image

Images don’t display unless the Image tab has focus. Same for Timeline widget.

## Timeline

Currently the First Visit band is hard coded. This information needs to be retrieved using first date from Mary’s OpenVista Visits.

Click on an appointment in the Timeline, and bring up details of the visit. (This info from Mary.), see above on example of how to do this.

Keyboard keeps popping up on iPad when click Timeline. Annoying.

## Modal Windows

Since moving from iFrames, the modal windows do not always work well. Needs testing.

All modal windows are called Editor Tab.

## Associate Patient

We need a mechanism to add a new patient to our system, when they exist currently in VistA. Dependant upon Patient cache functionality. (More documentation here.)

## Add an Image to database

Need a mechanism to add an image to our system, so that the image can be tagged with meta data.

## Listing of hData and OpenVista Data

As these are not patient specific, they shouldn’t be saved when we save patient data.

## Templates

Need to add template functionality. So that bring up a range of widgets at start for a new patient.

## Vital Signs Charts

Use the vital signs data from Mary to create charts.

## Other Possible OpenVista Data

* Lab Visits.
* Lab results/ labs pending?

## Header Information

Make sure that the header info is populated. (Vitals, Patient data, Problem list, etc,.)

## Allow for Entering of Problem List data

## Licenses

Update Ray on the list of licenses of plugins that we are currently using.

## iPad

Sometimes a Tab will appear with a concatenated list of titles. Sporadic though. Hard to recreate.