

# QDF Interfaces

## Table of contents

1 Abstract.....	2
2 Introduction.....	2
3 Problem Description.....	2
4 Details.....	3
4.1 Source XML Documents.....	3
4.2 Developing the Menu.....	3
4.3 Developing the Form.....	4
4.4 Developing the Portlets.....	9
5 Conclusion.....	12
6 References.....	12

## 1. Abstract

This paper discusses dynamically generating user interfaces from a Query Description Formatted (QDF) document for use within the Gridsphere-based CHRONOS portal. It gives a brief introduction into the problem domain and discusses the implemented solution. Some familiarity with XML, XSL, CSS, Javascript, and portals/portlets is assumed.

## 2. Introduction

The Query Description Format (QDF) is a technology that is being developed by the CHRONOS project. It is an XML dialect for describing queries into or services provided by a Service Oriented Architecture (SOA) system. The goal is to provide a mechanism for describing queries and services to both the back end systems, which execute the queries, and the front end systems, which provide interfaces to the users, in a single format. This way changes made can instantly be propagated to the back end systems and reflected in user interfaces.

Being able to dynamically generate user interfaces from the QDF document greatly eases the burden on the system developers. They no longer have to worry about writing a custom user interface into the service code that they develop. Nor do they have to worry about maintaining that user interface code. Instead they can focus on providing the end user with the services and queries that he or she requires.

Dynamically-generated interfaces are a boon to the end user as well. The users can be assured that they are seeing the most up to date version of the interface, built directly from the query specification. It also means that all the interfaces will have a consistent look and feel.

## 3. Problem Description

We wanted to dynamically generate usable forms-based HTML interfaces from the [CHRONOS QDF document](http://services.chronos.org/qdf/qdf.xml) (<http://services.chronos.org/qdf/qdf.xml>). The interfaces would then be integrated into one or more portlets for use on our Gridsphere-based portal.

The backend was already implemented. We had a system (servlet) that interprets the QDF document and performs the queries described in it. This was implemented as a REST-based system e.g. by specially crafting a URL you can invoke any of the queries in the QDF document ([Example that queries a database and returns the results as an HTML form](http://services.chronos.org/qdf/query/SQL-TIMESCALE-00000001) (<http://services.chronos.org/qdf/query/SQL-TIMESCALE-00000001>)). We pull the search parameters from the GET or POST request so an HTML form works quite nicely as an interface.

## 4. Details

This section will describe in detail the various steps that I took in building the interface. It is divided into subsections covering each of the major components--the menu, the forms, and the portlets.

### 4.1. Source XML Documents

- [QDF document](http://services.chronos.org/qdf/qdf.xml) (<http://services.chronos.org/qdf/qdf.xml>) , ([schema](http://services.chronos.org/qdf/qdf.xsd) (<http://services.chronos.org/qdf/qdf.xsd>) )
- [Types document](http://services.chronos.org/qdf/types.xml) (<http://services.chronos.org/qdf/types.xml>) , ([schema](http://services.chronos.org/qdf/types.xsd) (<http://services.chronos.org/qdf/types.xsd>) )

The interfaces will be built from the QDF document. We will consult the Types document at transform time for things like descriptions and examples for the drop down lists.

### 4.2. Developing the Menu

- [Menu XSL](http://services.chronos.org/qdf/xslt/menu.xsl) (<http://services.chronos.org/qdf/xslt/menu.xsl>)
- [Menu CSS](http://services.chronos.org/qdf/css/menu.css) (<http://services.chronos.org/qdf/css/menu.css>)
- [Menu JS](http://services.chronos.org/qdf/scripts/menu.js) (<http://services.chronos.org/qdf/scripts/menu.js>)
- [DHTML JS](http://services.chronos.org/qdf/scripts/dhtml.js) (<http://services.chronos.org/qdf/scripts/dhtml.js>)

The first component I worked on was the menu system. The goal for the menu was to show a brief summary of what was in the QDF document and allow the user to navigate to different queries.

There is nothing too revolutionary here. The XSL just pulls out the titles and descriptions of each query and builds a list. I do a few things to make the menu as usable as possible for the user.

The first is grouping and sorting. Each query definition has one or more <label> tags. These are used to indicate which queries are similar to each other. The contents of the tag is just a free form string, but we use certain conventions when labeling. If the query is developed for or used by a specific application, then I'll use the application name as one of the labels. If the query is against a database, I'll use the database name as one of the labels. And queries can have multiple labels so we don't have to worry about coming up with a single classification. When I build the list, I group queries by their labels and then sort them alphabetically within their groups. If a query has more than one label, then it will show up more than once in the list.

The second thing I do for usability is to hide the descriptions of the queries by default. Most users probably don't care about reading the descriptions when they are browsing, and leaving

them visible triples the length of the menu. To make HTML elements hidden, just set the style *display: none*. You can do this directly on the tag:

#### menu.html

```
<div class="hidden" style="display: none">
  Some content here to hide. The user won't see me as
  long as my style is set to display: none.
</div>
```

or you can do it with CSS:

#### menu.css

```
div.hidden { display: none }
```

#### menu.html

```
<div class="hidden">
  Some content here to hide. The user won't see me as
  long as my style is set to display: none.
</div>
```

Now that the element is hidden, we can make it display by setting the style *display: block*. I use a little javascript to display or hide the descriptions. It just iterates over all of the elements in the page and if the element is of a specific class ('description' in this case) I set the display property appropriately. You can find this function and other useful DHTML functions in `dhtml.js` linked above.

The third usability thing I do is provide a drop down list of all the labels and a field to type in a word or two. By selecting a label in the drop down list or typing in a keyword, the user can limit what queries are displayed in the list. This will be especially useful as the number of queries increases.

Finally, I put the whole menu into a fixed size div (width: XXXpx; height: YYYpx;) and set the style *overflow: auto*. When the menu gets bigger than the width or height I set, scrollbars are automatically added. This is nice so the user can scroll just the menu instead of the whole screen.

In the **Developing the Portlets** section I'll discuss how the menu interacts with the portlet to create a dynamic experience.

### 4.3. Developing the Form

- [Form XSL](http://services.chronos.org/qdf/xslt/form.xsl) (http://services.chronos.org/qdf/xslt/form.xsl)
- [Form CSS](http://services.chronos.org/qdf/css/form.css) (http://services.chronos.org/qdf/css/form.css)
- [Form JS](http://services.chronos.org/qdf/scripts/form.js) (http://services.chronos.org/qdf/scripts/form.js)

## QDF Interfaces

- [DHTML JS](http://services.chronos.org/qdf/scripts/dhtml.js) (<http://services.chronos.org/qdf/scripts/dhtml.js>)

The form XSL is much more involved and interesting than the menu XSL. There's too much to go over everything but I'll highlight some of the more interesting and useful features.

### Form Usability Tricks

One of the first things users notice when using the forms is the feedback that the form gives you. When you select a field, that field's border changes to indicate that you're ready to enter text. Required fields are marked by red circles. When you enter a value, the red circle becomes a green check mark. If you tab through a field that is required without entering a value, the red circle becomes a little exclamation point. I wish I could say I thought of these spiffy things myself but they are from Simon Willison's article "Simple Tricks for More Usable Forms" out at [SitePoint](http://www.sitepoint.com/article/simple-tricks-usable-forms) (<http://www.sitepoint.com/article/simple-tricks-usable-forms>) . It is a good read for anyone doing web development.

### Tooltips

I'm sure everyone knows what tooltips are. I decided to use them here because they give a more "application-like" feel and because they can display additional information without cluttering up the interface. I'm using Walter Zorn's [Tooltip library](http://www.walterzorn.com/tooltip/tooltip_e.htm) ([http://www.walterzorn.com/tooltip/tooltip\\_e.htm](http://www.walterzorn.com/tooltip/tooltip_e.htm)) .

I placed tooltips on each of the fields in the form. They provide some helpful information as to what is supposed to go in each field. The text for each tooltip is built from the description of the parameter in the QDF document and the description of the type in the Types document. Here's the source XML from the QDF and Types documents:

#### qdf.xml

```
<param name="species" type="Species" label="Species" required="true">
  The species value to conduct search.
</param>
```

#### types.xml

```
<type name="Species">
  <description>A string specifying the species name of a
  taxa.</description>
  <examples>
    <value>TITAN</value>
    <value>MITRA</value>
    <value>CYCLOSTOMUS</value>
    <value>TOM%</value>
  </examples>
</type>
```

And here is the XSL code that transforms the XML into text for the tooltip:

#### form.xsl

```

<xsl:variable name="type">
  <xsl:value-of select="normalize-space(@type)"/>
</xsl:variable>

<!-- this builds up our tooltip -->
<xsl:variable name="desc">
  <xsl:text>&lt;b&gt;Description:&lt;/b&gt;&lt;br/&gt;</xsl:text>
  <xsl:value-of select="normalize-space(.)"/>
  <xsl:text>&lt;br/&gt; &lt;br/&gt; &lt;b&gt;Type:&lt;/b&gt;
</xsl:text>
  <xsl:value-of select="$type"/>
  <xsl:text>&lt;br/&gt;</xsl:text>
  <xsl:value-of
    select="normalize-space(
      document('http://services.chronos.org/qdf/types.xml')
        /types/type[@name = $type]/description)"/>
</xsl:variable>

```

This whole chunk is in a template that processes each `<param>` tag defined in the query definition. The first couple of lines pulls out the type attribute from the `<param>` tag and saves it for use later. Then I start building up the text for the tooltip. I wanted to embed some HTML tags into the tooltip so I had to escape the in the XML. Alternatively, I could have used a `<![CDATA[ ]]>` around them. The first line says I want '**Description:**' in bold on it's own line. Then include the text of the `<param>` tag. Next include two line breaks and '**Type:**' in bold. After that include the type of the `<param>` tag and a line break. The last line says open up the XML document at 'http://services.chronos.org/qdf/types.xml' (the types document) and pull in the description of the type from that. The `document()` function allows us to pull in content from multiple XML documents.

Using the XSL above on the XML from the QDF and Types documents, we get a tooltip that looks like this:

### Tooltip Image

#### Building Drop Down Lists

After presenting initial prototypes of the interface to the people that will actually be using it, the most frequent comment that I got was: "I see the field and I see the tooltip but I still don't know what to type into it. Can you provide some examples of valid entries?" And this was coming from scientists, people who have knowledge in the problem domain. If they couldn't guess what was a valid entry, then there was no chance for the lay person.

I looked over the interface and determined that there were two major cases: fields that had a well defined (finite) set of valid entries and fields that could receive arbitrary input. The first case is akin to entering your state/province/country into a form. There is no reason to let the user type anything in because a drop down list can present all of the valid choices. The drop down will also ensure that the user doesn't mistype. For the second case, because there is no finite set of acceptable answers, we provide both a text input and a drop down list of example

## QDF Interfaces

entries. This is helpful to the user so they can figure out if they are supposed to be entering a number or a string or whatever.

To support this functionality, I added a section to each of the entries in the Types document. For the first case, I added a <set> tag to the type definition:

### types.xml

```
<type name="InfoTarget">
  <description>A string specifying the target of a System Info
query.</description>
  <set>
    <value>all</value>
    <value>databases</value>
    <value>formatters</value>
    <value>serializers</value>
    <value>queries</value>
  </set>
</type>

<type name="TimeScaleName">
  <description>A string specifying the name of a time scale in the
CHRONOS System.</description>
  <set
url="http://services.chronos.org/qdf/query/SQL-TIMESCALE-00000001?serializeAs=wrs"
nodeName="columnValue"/>
</type>
```

To accommodate the second case, I added an <examples> tag to the type definition:

### types.xml

```
<type name="Species">
  <description>A string specifying the species name of a
taxa.</description>
  <examples>
    <value>TITAN</value>
    <value>MITRA</value>
    <value>CYCLOSTOMUS</value>
    <value>TOM%</value>
  </examples>
</type>
```

With that in place, I can use the same document() trick I used in building the tooltips to pull in the various values and build a drop down list. Here's the XSL snippet that does it:

### form.xsl

```
<xsl:when
test="document('http://services.chronos.org/qdf/types.xml')/types/type[@name
= $type]/set/value">
```

```

        <!-- the types defined a set with values so pull in all the values
for a drop down -->
        <select name="{@name}" id="{ $id}" onchange="setLabelClass('{ $id}',
'completed');" >
            <option name="" value="">Select...</option>
            <xsl:for-each
select="document('http://services.chronos.org/qdf/types.xml')
/types/type[@name = $type]/set/value">
                <option name="{.}" value="{.}"><xsl:value-of
select="."/;></option>
            </xsl:for-each>
        </select>
    </xsl:when>

    <xsl:when
        test="document('http://services.chronos.org/qdf/types.xml')
/types/type[@name = $type]/set[not(@url = '')]">
        <!-- the types defined a set with a url so pull in all the
values that match nodeName for a drop down -->
        <xsl:variable name="url">
            <xsl:value-of
select="document('http://services.chronos.org/qdf/types.xml')
/types/type[@name = $type]/set/@url"/>
        </xsl:variable>
        <xsl:variable name="xpath">
            <xsl:value-of
select="document('http://services.chronos.org/qdf/types.xml')
/types/type[@name = $type]/set/@nodeName"/>
        </xsl:variable>

        <select name="{@name}" id="{ $id}" onchange="setLabelClass('{ $id}',
'completed');" >
            <option>Select...</option>
            <xsl:for-each select="document($url)//*[local-name() =
$xpath]">
                <option name="{.}" value="{.}"><xsl:value-of
select="."/;></option>
            </xsl:for-each>
        </select>
    </xsl:when>

    <xsl:when
test="document('http://services.chronos.org/qdf/types.xml')/types/type[@name
= $type]/examples/value">
        <!-- the types defined some examples so pull in all the values for
a drop down -->
        <input type="text" name="{@name}" id="{ $id}" class="{ $req}"
onblur="{ $check}"
            onchange="setLabelClass('{ $id}', 'completed');"
style="margin-right: 5px;"/>
        <select name="{@name}-example" id="{ $id}-example"
            onchange="document.getElementById('{ $id}').value =
this.options[this.selectedIndex].value;
            setLabelClass('{ $id}', 'completed');" >

```

## QDF Interfaces

```
        <option>Examples...</option>
        <xsl:for-each
select="document('http://services.chronos.org/qdf/types.xml')
        /types/type[@name = $type]/examples/value">
        <option name="{.}" value="{.}"><xsl:value-of
select="."/></option>
        </xsl:for-each>
    </select>
</xsl:when>

    <xsl:otherwise>
        <!-- nothing defined in the types so the user is on his own -->
        <input type="text" name="{@name}" id="{@name}" class="{ $req}"
            onblur="{ $check}" onchange="setLabelClass('{ $id}',
'completed');" />
    </xsl:otherwise>
```

It looks long and complicated but it really isn't too bad. It is long because we have to handle four cases: a <set> with <value>s, <set> that defines a URL, <examples> with <value>s, and nothing defined. The most interesting case is the second one: a <set> that defines a URL. It allows me to pull in data from yet another XML source. Humorously enough, in the cases that I use the <set> that defines a URL, I am actually calling queries defined in the QDF document. Oh Recursion! :)

### IFrame Goodness

One thing you may notice while using the interface is that even though I am using an HTML form, the form never gets submitted (the page doesn't refresh when you hit the submit button). Instead, when you hit the submit button, the iframe in the page displays the results. This works because the backend QDF system is REST-based. I can build a URL to execute any query in the QDF document. So I use a little bit of Javascript code to check that the user has filled in all the required fields and then build the REST URL to the desired results.

I think this is a nice touch. It keeps the user in the search mentality by not breaking things up with a page refresh. The user can quickly tweak the results by changing the parameters and re-submitting without waiting for the server to re-render the page. It also means that the user's entered values don't get lost. A final advantage to this is that it saves the server from having to perform a needless transform on the QDF document.

## 4.4. Developing the Portlets

- [XSLTPortlet.java](#)
- [QDFMenuPortlet.java](#)
- [QDFAppPortlet.java](#)

To plug the interfaces into our Gridsphere Portal I reused some portlet code that I had developed for another project. It is a generic portlet that takes a source XML file, an optional XSL file, and a JSP file. It then transforms the XML file with the XSL and forwards the

results to the JSP. I had previously used the portlet to transform RSS feeds into HTML for displaying on the portal.

The main motivation behind the XSLTPortlet was to eliminate making a separate class file for each RSS feed we wanted to display. I did this by making a sufficiently generic portlet that got its source, XSLT, and JSP parameters from its portlet definition in the portlet.xml file. This way, if I wanted to add a new portlet for an RSS feed, I just would add another definition in the portlet.xml file and change the source.

Once that was working well, I thought that if I added the ability to change the source, XSLT, or JSP I could increase the usefulness of this generic portlet. So I added a little bit of code to the doView() and processAction() methods to check if a source, XSL, or JSP parameter was passed in. If it was I updated my portlet-scoped session variables and used the new updated parameters. This was useful because I could dynamically change which stylesheet was used e.g. I could switch between the *menu.xsl* view of the QDF document and the *form.xsl* view of the document by just building an action URL to submit back to the portlet with a *xsl=form.xsl* parameter

This was useful but I was having trouble because I really wanted to be able to build my URLs inside the XSL transform. Obviously I couldn't use the Gridsphere UI tags within the transform. My solution was to attach various useful values like the action URL, render URL, and CID to the request as attributes. Then I could do stuff with those strings in the JSP. The transformed XML was coming back as one big, long string attached to the request as well. Since I was dealing with strings I thought the best way to build my URLs within the XSL was to put some text token in their place and replace the tokens with the actual values when the request got to the JSP. So that's what I did. In the XSL I used tokens like **\$ACTION\_URL**, **\$RENDER\_URL**, and **\$CID** to indicate where I wanted those values to be. And then when everything got to the JSP, I just did a `results.replaceAll("$ACTION_URL$", actionURL)`. Viola! Some examples are below:

#### menu.xsl

```
<div class="title">
  <a
href="$ACTION_URL$&target=http://services.chronos.org/qdf/query/qdf%3Ftarget%3D{id}">
    <xsl:value-of select="title"/>
  </a>
</div>
```

#### qdf.jsp

```
result = result.replaceAll("\\$RENDER_URL\\$", rURL);
result = result.replaceAll("\\$ACTION_URL\\$", aURL);
result = result.replaceAll("\\$CID\\$", cid);
```

## QDF Interfaces

```
out.write(result);
```

Not the most elegant solution but it works. And surprisingly well.

The final thing I had to do with the portlets was get them to talk to each other. I knew I was going to have two portlets on the page. One that used the QDF document and the menu.xsl transform I developed. The other would use the QDF and the form.xsl transform. I needed a way for the menu portlet to indicate which query the form portlet should display. The solution I decided on was to use an application-scoped session attribute. I knew that both portlets could read and write the attributes that were application-scoped. So I extended the XSLTPortlet for the menu portlet so that when it received a target parameter, it would place that into the session:

### QDFMenuPortlet.java

```
public class QDFMenuPortlet extends XSLTPortlet {
    protected void doView(RenderRequest arg0, RenderResponse arg1)
        throws PortletException, IOException {
        // put targets into the session
        PortletSession session = arg0.getPortletSession();
        String target = arg0.getParameter("target");
        if (target != null) {
            session.setAttribute("target", target,
PortletSession.APPLICATION_SCOPE);
        }
        super.doView(arg0, arg1);
    }
    public void processAction(ActionRequest arg0, ActionResponse arg1)
        throws PortletException, IOException {
        // put targets into the session
        PortletSession session = arg0.getPortletSession();
        String target = arg0.getParameter("target");
        if (target != null) {
            session.setAttribute("target", target,
PortletSession.APPLICATION_SCOPE);
        }
        super.processAction(arg0, arg1);
    }
}
```

And likewise, I had the form portlet look into the session for a target attribute and it would override the source:

### QDFAppPortlet.java

```

public class QDFAppPortlet extends XSLTPortlet {
    protected void doView(RenderRequest arg0, RenderResponse arg1)
        throws PortletException, IOException {
        // retrieve targets from the session
        PortletSession session = arg0.getPortletSession();
        String target = (String)session.getAttribute("target",
PortletSession.APPLICATION_SCOPE);
        if (target != null) {
            URL s = makeURL(target);
            session.setAttribute("source", s);
        }
        super.doView(arg0, arg1);
    }

    public void processAction(ActionRequest arg0, ActionResponse arg1)
        throws PortletException, IOException {
        // retrieve targets from the session
        PortletSession session = arg0.getPortletSession();
        String target = (String)session.getAttribute("target",
PortletSession.APPLICATION_SCOPE);
        if (target != null) {
            URL s = makeURL(target);
            session.setAttribute("source", s);
        }
        super.processAction(arg0, arg1);
    }
}

```

## 5. Conclusion

Whew! We made it to the end. I hope I didn't bore you to death with my ramblings, and perhaps you even learned a trick or two. The QDF project, both the backend and the frontend, are under constant development. We recently added the ability to expose the queries in the QDF as both REST services and also SOAP services. Likewise, the interfaces I discussed here are constantly being improved. We are sending out a formal usability survey to our users in early March 2005 and hope to get valuable feedback. If you have any comments, suggestions, or questions feel free to contact me at [jareed@iastate.edu](mailto:jareed@iastate.edu) (<mailto:jareed@iastate.edu>) .

## 6. References

- [CHRONOS Portal](http://portal.chronos.org/gridsphere/gridsphere?cid=search_datasearch)  
([http://portal.chronos.org/gridsphere/gridsphere?cid=search\\_datasearch](http://portal.chronos.org/gridsphere/gridsphere?cid=search_datasearch))
- [Gridsphere](http://www.gridsphere.org/) (<http://www.gridsphere.org/>)

## QDF Interfaces

- [QDF document](http://services.chronos.org/qdf/qdf.xml) (<http://services.chronos.org/qdf/qdf.xml>) , [QDF schema](http://services.chronos.org/qdf/qdf.xsd) (<http://services.chronos.org/qdf/qdf.xsd>)
- [Types document](http://services.chronos.org/qdf/types.xml) (<http://services.chronos.org/qdf/types.xml>) , [Types schema](http://services.chronos.org/qdf/types.xsd) (<http://services.chronos.org/qdf/types.xsd>)
- [XSL directory](http://services.chronos.org/qdf/xslt/) (<http://services.chronos.org/qdf/xslt/>)
- [CSS directory](http://services.chronos.org/qdf/css/) (<http://services.chronos.org/qdf/css/>)
- [Scripts directory](http://services.chronos.org/qdf/scripts/) (<http://services.chronos.org/qdf/scripts/>)
- [QDF Javadocs](http://services.chronos.org/qdf/docs/javadocs/) (<http://services.chronos.org/qdf/docs/javadocs/>)
- [Simple Tricks for More Usable Forms](http://www.sitepoint.com/article/simple-tricks-usable-forms) (<http://www.sitepoint.com/article/simple-tricks-usable-forms>)
- [Tooltip Library](http://www.walterzorn.com/tooltip/tooltip_e.htm) ([http://www.walterzorn.com/tooltip/tooltip\\_e.htm](http://www.walterzorn.com/tooltip/tooltip_e.htm))