

HistoryBrowser

Virginia Center for Digital History

Overview

Bill Ferster

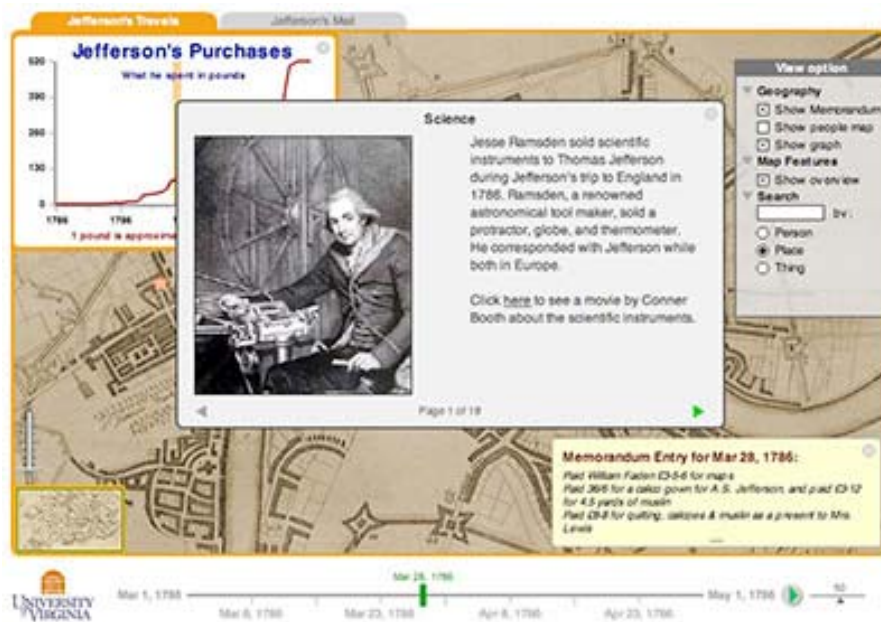
8/15/09

The HistoryBrowser allows users to interactively browse historical events in a number of ways. It is a *generative browser*, allowing users to not only view preset collections of events, but to construct their own views of the events based on selected criteria. The HistoryBrowser makes it easy to construct complex queries about historical events, weaving maps, timelines, and data visualizations to encourage insight.

The HistoryBrowser is a tabbed-based collection of views of historical events and data that can be interactively shown within a time period using the timeline tool. Views can contain event descriptions, primary source documents and imagery, maps, digital movies and audio, animations, charts and graphs of historical data.

Each view represents the result of choices of what to show in that view. The views can show events that match certain criteria, ranging from “show all events in Antioch, Virginia” to a sophisticated query such as “show all events where Jefferson bought more than 3 trees from 1796 to 1820, but not ones from Thomas Mayne.”

These views can be fixed for demonstration purposes, or left open, for people to explore various relations between the elements provided allowing for both purposeful and serendipitous discovery of complex interrelations.



Resources Supported

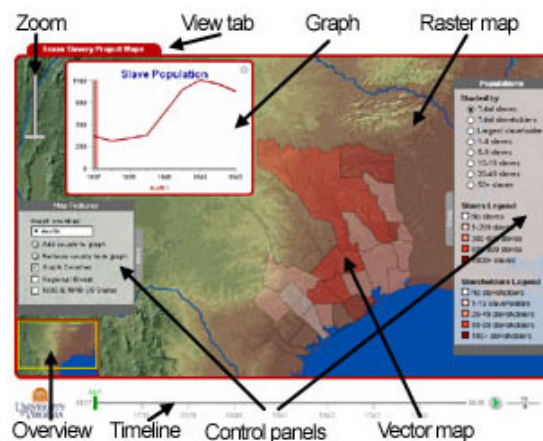
There are three basic kinds of information (resources) the HistoryBrowser can display and search for:

1. **Maps** - The HistoryBrowser contains a fully interactive geographic information viewer to display three basic kinds of maps. 1) Scans of historical maps, 2) vector-based maps from GIS systems such as arcGIS, and 3) maps delivered from Internet-based web services, such as Google Maps, or any combination of the three. All maps can be easily panned and zoomed, with an option to see an overview inset. Shape data can come from specified tables in a SQL/mySQL database, a link to an XML file, or from Internet-based web service.
2. **Data** - A rich array of historical data can be imported into the HistoryBrowser from a database as a table. This data can be displayed as a layer on a map, shown as a chart, table, or graphic element, and most importantly, used to include or omit events in the views. Data can come from specified tables in a SQL/mySQL database, a link to an XML file, or from Internet-based web service.
3. **Images/Movies** - Digitized images of primary source documents from digital archives can be displayed and integrated into maps, animations and other visualizations. These images can be JPEG, GIF or PNG formats, and can be dynamically sized and positioned, movies on YouTube; and Flash movies and animations.

View Components

An unlimited number of views can be constructed from these three basic resource elements. Views can be interactive, enabling users to change how the various resources interact, or static, allowing for a didactic interpretation of the events.

The infrastructure for a view contains a procedural description of how the information is displayed with conditional comparisons and loops so that very sophisticated queries can be performed. These queries are easily constructed using pull down menus for the various options desired indicating the relationships between the various resources.



1. **Control Panels** - Each View has its own pull-out area docked to a side of the screen that can be expanded or collapsed as needed and contains a number of collapsible check boxes to toggle on and off various features of the map, such as data overlays, roads, town names, etc. Various map features, such as the overview navigation insert and map legend can be turned on and off here as well, but assigning some "Glue" to be activated on clicking. Control panels can contain:
 - Radio Buttons
 - Check Boxes
 - Combo Boxes
 - Slider Controls
 - Text Input Boxes
 - Buttons
 - Headers and Legends
2. **Timelines** - Each view can have its own timeline that can control the temporal aspect of the project. Sliding the cursor changes the view's date, which in turn can change the way in which information is displayed if it is time dependent.
3. **Animation Players** - The current time on the timeline can be animated over time, using a player control, allowing the project to animate any time dependent elements from any point on the timeline to another.
4. **Zoomers and Overviews** – The screen can be controlled by a zoom slider and/or a small overview inset that facilitates panning through the screen.
5. **Graphs** – Various types of graphs (line, bar pie, scatter, etc.) can be drawn using dynamically generated data base on data from XML or SQL databases, time from timeline, settings of control panel items, or any combination of them.
6. **Tables and Text Displays** – Dialog-box based tables and text displays can be drawn using dynamically generated data base on data from XML or SQL databases, time from timeline, settings of control panel items, or any combination of them.
7. **Paths** – A series of positions on the screen (specified by pixels or latitude, longitude if a map) can be defined to appear at particular times. Each position can be marked by an icon, shape or image file. Clicking on the position can call up a web page draw graphical elements, or popup window showing some information. Lines can be drawn to connect these positions.
8. **Radial Maps** – A path can be arrange in a radial concept map format to help visualize relationships between objects shown.

HistoryBrowser XML Guide

The HistoryBrowser is an empty vessel for interacting with historical information. Its entire functionality and “look and feel” is controlled by an XML data structure. This flexibility will allow it to be effectively used in a wide variety of projects, while still maintaining a common internal structure.

The topmost level on the hierarchy is the project. The project contains any number of views. This will allow for multiple interactive representations to be shown simultaneously, making for easy comparisons, with a shared timeline for control.

Each browser may contain any number of views. These views are represented as tabbed areas on the screen. Clicking on any of the tabs will bring up a different view. Each view contains descriptors of the *resource* to display, or use as data to change the display. There is one timeline for each view, which makes it easy to set the temporal aspect of that view.

- **Resource**

The resource could be a *map*, some *media*, a table of *data*, or a *graphic*. Each resource item contains a query instruction as how to find the data for that resource. For example, a map may be a URL to a bitmap, a SQL query for a collection of shape files in a database, or a URL to an online web service, such as Google maps. A data table from a SQL database, from an XML file or a web service can be similarly used.

A number of objects, such as timelines, charts, tables, graphs, etc. can be displayed on the screen using resources and glue defined below.

- **Glue**

Glue (The General Language to Unite Events) is a procedural description of how the various resource elements connect with one another and are displayed. The HistoryBrowser knows how to render a number of types of resource, such as tables, charts, text area, movies, audio clips, vector and raster maps, and the Glue language contains elements to cause them to display. The Project and ControlPanel rely on Glue to know how to display the views and sub-views.

Glue also contains elements for linking user-generated actions, such as clicking on the screen with actions. Glue also provides an opportunity to calculate tables and fields in resources based on a simple script in the tag. Many common types of operation can be defined between these elements, so that the HistoryBrowser is able to relate rich data relationships between them and visualize them on a special and temporal basis.

- **ControlPanel**

Each view contains a control panel that can be populated with a number of interface items, such as checkboxes, radio buttons sliders and header elements. This panel can be docked anywhere in the view or be free-floating. It can be always present, or opened and closed like a drawer

- **Containers**

Containers are collections of objects used to put information on the screen on top of a resource, such as an image or a map. There are currently three types of a container: A Path creates trails and navigation objects, a CMap makes concept maps, and a Canvas defines a scrollable area to contain images.

Projects

The **project** defines the top-most mode of a HistoryBrowser project. The **host** attribute sets URL where the .swf file resides. The **frame** defines the boundaries of the view panels and the **tab** area combined. The **textFormat** defined will serve as the default format used for all views if not re-defined.

The **logo** defines a bitmap to use across all views, and is typically below the area defined by the **frame**. The **tab** defines the height, width, and colors of the tabs, if any. The **view** tags provide the content for all of the views within the project. There is no limit to the number other than what will fit horizontally as tabs.

```
<project title="name" host="url_of_host" >                                // Topmost level
  <textFormat> textformat </textFormat>                                // default text attributes
  <frame> frame </frame>                                              // frame of project
  <logo top="pixels" left="pixels" source="url" />                       // logo
  <tab                                                                    // defines tabs
    onCol="0xrgb" offCol="0xrgb"                                         // color of on/off tabs
    onTextCol="0xrgb" offTextCol="0xrgb"                                 // text color tabs
    hgt="pixels" wid="pixels"                                           // size of tabs
    curView="number" />                                                // starting tab to see
  <view > view1-n* </view> ...                                        // tabbed view(s)
</project>
```

Views

Each tab in the project contains a **view**. The **view** contains elements that are displayed on the view's **screen**. The **textFormat** will use the **project**'s text formatting as it's basis and elements can be over-ridden. **Resources** such as maps, images and data are loaded for display. **controlPanels** are set up to provide user interface controls. The scope of any **view** is itself, meaning each **view** is an island unto itself.

Almost all of the other elements that make up a project are within one or more view elements and provide the top-level navigation control fro your project.

Views can also be invisible and not associated with any particular tab. By setting the *visible* attribute to "true" and giving it an *id*, you can use GLUE to cause a view to show within the currently active tab's screen space. See the **setview()** GLUE element and the section on the cookbook section for more information on this very powerful option.

*NOTES ABOUT FORMAT OF XML

Italics indicate link to another XML object
... indicates there may be multiples of these objects
underlined option in [multiple | choices] indicates default

```

<view id="id" title="name" >
    <textFormat> textformat </textFormat>
    <resource> resource </resource> ...
    <controlPanel> controlPanel </controlPanel> ...
    <timeline> timeline </timeline>
    <glue> glue </glue>...
    <path> path </path> ...
    <cmap> conceptMap </cmap> ...
    <zoomControl
        top="pixels" left="pixels"
        def="number"
        max="number" />
    <overview
        docking="[ botLeft | topLeft | botRight | botLeft ]*" // docking
        wid="pixels" boxCol="0xrgb:0xffff00" />
        def="[ true | false ]" // show at startup?
        src="inset.jpg" />
        visible="[ true | false ]" // visible or free-floating view
</view>

```

// Tabbed views of data

```

// overridden text attributes
// resources(s) for this view
// control panel(s) for view
// timeline for this view
// connection mapping
// path(s) for this view
// concept maps for this view
// zoom control for view
// position (omit to dock it)
// starting value
// max zoom (% /100)
// overview navigation control
// width and control box color
// inset map image

```

Control Panels

ControlPanels provide a dialogbox-like means for setting parameters of the screen. These parameters can be set using **items** such as check boxes, radio buttons, combo selection boxes, sliders, text input, and buttons to cause some sort of action. Items typically cause some action by adding an id of a **glue** tag to call when they are changed or clicked.

The **frame** specifies the size, position and color of the **controlPanel**. The *docking* attribute in the **frame** can be *top*, *left*, *bottom*, *right*, or *float*. The first 4 cause the **controlPanel** to “stick” to that side of the screen, ignoring the positioning parameters (top & left) set in the **frame**. The setting the *docking* to “float” will allow the **controlPanel** to be positioned anywhere on the screen as dictated by the “top” and “left” attributes.

The **textFormat** will use the **view**'s text formatting as it's basis and elements can be over-riden. The title sets the name in the **controlPanel** 's bar. Setting the *closable* attribute adds a handle to collapse the panel into the screen's frame. The *open* attribute sets the default status at startup of the **controlPanel** either open or closed.

```
<controlPanel title="name" >                                     // Control panel for views
  closable="[ true | false ]"                                     // enable panel closing tab
  open="[ true | false ]"                                        // panel closing startup status
  <frame> frame </frame>                                         // box of control panel
  <textFormat> textformat </textFormat>                          // default text attributes
  <item id="name" > ...                                         // each line in the panel
    type="[ checkbox | radio | slider | textbox |               // item type (buton is not a typo!)
           header | legend | buton | buttonbar |
           line | text | combobox | half ]"
    def="value"                                                 // default value/state of item
    title="name"                                               // name of item to show
    bold="[ true | false ]" italic="[ true | false ]"         // is text bold or italic
    <glue> glue </glue> *                                       // glue object to run if clicked
    min="value" max="value"                                     // min/max values (slider only)
  </item>
</controlPanel>
```

The **items** form the active portion of the **controlPanel** as a running list of objects. They are drawn using the current font settings, but the *bold* and *italic* can be over-riden on a per-item basis. The current *leading* in the **textFormat** item determines how far they are spaced vertically. If there are more items than fit vertically, a new column is started.

Radio items act a a collective group, allowing only one radio on at a time. Header items will collapse items below it (until the next header item). Legends provide a box colored by the def attribute, and are stacked from the bottom of the **controlPanel** up. Setting the glue of a checkbox item to “legend” will cause the legends to collapse if checked

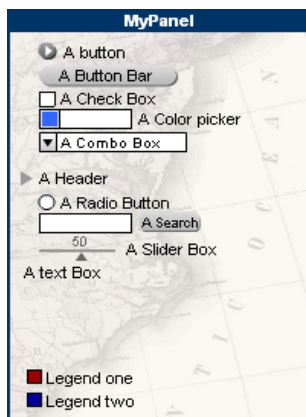
You can change control panel items dynamically by using the **menuitem()** GLUE method. You can also see what a given item's value is set to by using the item's id. For example, `status (*myCheckBox)` would print “1” if checked, and “0”, depending on the checked state of a **controlPanel item** named “myCheckBox.”

*built-in glue objects “legend” & “inset” toggle on/off if in a “checkbox”

Types of Control Panel Items

| | |
|------------------|--|
| buton | A round button that will trigger a glue method when clicked |
| buttonbar | A square button with the title written inside that will trigger a glue |
| checkbox | A checkbox with the title to the right that will trigger a glue method when clicked. |
| color | A color chip to choose a color from a set of choices, or type the RGB values |
| combobox | A combo box to choose between several choices |
| half | Used to add a half-space vertically (leading) to the list |
| header | An arrow control to collapse or expand the items the follow until another header |
| legend | Used to put a color choice when drawing legends |
| line | Draws a separator line |
| query | Adds a query line (if: something equals value) |
| radio | A radio button, of which only one is active in a contiguous group of them |
| search | A text input box with a search button bar attached |
| slider | A horizontal slider to set the value from 0-100 |
| text | Displays a line of text |
| textbox | A text input box |

Here is an example of a control panel with many of the items added:



```
<controlpanel title="MyPanel">
  <textformat leading="16" />
  <frame docking="left" alpha="75" wid="200" hgt="300" />
  <item title="A button" type="buton" />
  <item type="buttonbar" title="A Button Bar" />
  <item title="A Check Box" type="checkbox glue="myGlue" />
  <item title="A Color picker" type="color" />
  <item title="Yes|No|Maybe" type="combobox" />
  <item title="A Half Line" type="half" />
  <item title="A Header" type="header" />
  <item type="legend" def="0x990000" title="Legend one" />
  <item type="legend" def="0x000099" title="Legend two" />
  <item title="A Radio Button" type="radio" />
  <item title="A Search Box" type="search" />
  <item title="A Slider Box" type="slider" />
  <item title="A text Box" type="text" />
</controlpanel>
```

Working with ComboBoxes

Combo boxes are useful items that contain a series of choices in a drop-down menu. When an option is selected, its name is set as the current value, just like a 0 or 1 is set with a checkbox. You set the values in the title attribute, separated by | marks like this:

```
<item title="Yes|No|Maybe" type="combobox" id="myCombo" glue="myGlue"/>
```

The glue can then get the currently selected option like this:

```
<glue id="myGlue">
  status(*myCombo)
</glue>
```

Timelines

The **timeline** tag will add a graphical timeline that will allow the user to set a time period along a horizontal timeline using a slider bar. A play button can be added to the timeline to animate the setting of the slider bar over time by setting *play* to “true.” Setting *speed* to “true” will put up a speed control aside the play button, with a default speed of 50%. That default can be over-ridden by setting the speed to some other number from 1-100.

The **frame** tag set’s the size and position of the timeline on the screen. The *wid* attribute sets the width, and *backCol* sets its color. The **textFormat** will use the **view**’s text formatting as its basis and elements can be over-ridden.

The *min*, *max* and *start* attributes define the minimum , maximum and starting dates for the timeline, and can be expressed as “year”, “month/year” or “day/month/year.” Dates can be formatted as years (1976), month/year (3/1856), day/month/year (3/7/1756), or full date (January 6, 1798).

The timespan can be divided by 4 major tick marks, whose length is set by *majorTick* and 3 minor tick marks set by *minorTick*. The tics can be place above, below and across the main timeline bar by setting *tickPos*. The *showValues* and *showMinorValues* determine if the tick mark’s time values are displayed.

Labels signifying specific dates can be added using a labels tag. What direction they emanate from the main bar is set by *pos*, and the distance away from the main bar is set by *offset*. Lines can connect the label to the main bar by setting the *lines* attribute. .The **textFormat** will use the **timeline**’s text formatting as its basis and elements can be over-ridden The actual entries are set using the **timelinelabels** Glue method (see GLUE section).

```
<timeline>
  <frame> frame </frame>
  <textFormat> textformat </textFormat>
  <timebar>
    play="[ true | false ]"
    speed="[ Number:50 | false ]"
    min="date" max="date"
    start="date"
    dateFormat="[ yr | mo/yr | dy/mo/yr | mo/dy/yr | mo.dy.yr]" // date format
    majorTick="pixels:0"
    minorTick="pixels:0"
    tickPos="[ top | mid | bot ]"
    numrTick="Number:4"
    showValues="[ true | false ]"
    showMinorValues="[ true | false ]"
    sliderDatePos="[ top | bot | none ]"
  <labels>
    pos="[ top | bot ]"
    lines="[ true | false ]"
    offset="number:8"
    <textFormat> textformat </textFormat>
  </labels>
</timeline>
```

// Timeline controller
// box of timeline
// specific text attributes
// for punctuated timeline
// show play button
// show play speed controller
// start and end dates
// starting slider dates
// date format
// major tick make length
// minor tick make length
// position of ticks rel to main bar
// number of ticks on timeline i
// show values of major ticks
// show values of minor ticks
// slider date’s position
// labels under timeline
// position of labels to main bar
// show lines from bar to labels
// distance of labels from bar
// specific text attributes

Punctuated Time Bar

You can add a segmented bar that will control the span of the overall timeline (called a punctuated timebar) by adding a **timebar** to the timeline. The timebar consists of a series of segments (must be contiguous!) that contain a starting and ending date within the overall *min* and *max* set in the timeline, and provides a way to make the timeline's span to be a portion of the full time being represented for better control and slowing animation of short events within the larger timeframe.

When a segment is clicked, the timeline's *min* and *max* settings are set to the segment's *start* and *end* attributes, making the timeline's span smaller. The screen is updated as if you had dragged the slider to the segment's start date. A button to make the timeline extend from the first segment to the last can be added by setting the *all* tag to true.

A glue tag can be specified to cause some glue to trigger when any segment is clicked. You can append parameters to the glue's name, which will be available as variables in the glue element.

For example, if we wanted to pass the start and end time to the glue, we would spec the glue as: `glue="newTime?2&8"`, the value of 2 would be set in `$$click` variable and the value of 8 would appear in the `$$param` variable within the `newTime` glue element.

```
<timebar>                                     // Punctuated timeline controller bar
  <segment>                                     // contiguous segments
    glue="glueID"                               // glue if clicked
    all="[ true | false ]"                     // show "show all" button
    onTextCol="0xrgb" offTextCol="0xrgb"       // color of active/inactive text
    onCol="0xrgb" offCol="0xrgb"               // color active/inactive segment
  </timebar>

<segment>                                       // Timebar segment
  start="date"                                  // segment starting date
  end="date"                                    // segment ending date
  title="string"                                // label of segment
</segment>
```

Resources

Resources contain information to be used by the HistoryBrowser. This information is most often a table of data, but can be an interactive vector map, text, images, animation, movies, audio, charts, and graphs. **Resources** are the raw material for the HistoryBrowser views. The **<resource>** tag in the project file provides a way to identify sources and provide *named access* to the data they contain. This access is useful because once they have been identified; we can refer to them by name later on using lines of Glue to easily create complex visualizations.

The various types of resources fall into four categories: 1) Image and Movie Resources, where images, movies and audio are loaded; 2) Data Resources, where numeric and/or string data organized into tables is loaded; 3) Map Resources load vector maps from **shapeData** imported from arcGIS and other GIS and graphics packages; and finally 4) Graph and Table Resources that add charts, graphs, tables and other pop-up graphics to display data and information.

Common Resource Attribute Tags

All **Resources** have some common tag attribute elements, such as the *id*, *preload*, *type*, *title*, and *onclick* tags. The *id* provide a way to uniquely identify the resource to other elements in a **view**. The scope of any resource is within the **view** it is contained by. If the *preload* tag is set, a spinning cursor will appear while that resource is being loaded. This is useful when the display is dependent on an element, such a basemap to load. The type defines the type of resource, i.e. map, image, data, etc. The *onclick* attribute defines a **glue** element to execute when right-mouse clicked by the user.

```
<resource id="name" >
    title="Name"
    type="resourceType"
    src="url"
    preload="[ true | false ]"
    onclick="glue"
</resource>
```

// Resource object
// short title of resource
// resource type
// web address of resource
// hold up start until it's loaded
// glue to run if clicked on

1. Image and Movie Resources

Image Resources allow you to add JPEG, GIF and PNG images from any valid URL provided in the *src* tag. These images are added directly to the view's screen (top-left corner by default, but can be anywhere, as set by *top* and *left* tags), where they can be panned and zoomed. Any number of images can be layer. Setting the *depth* to "topMost" will draw the image independent of any panning or zooming. Setting *wid* to non-zero, sets that image's width to that size.

```
<resource>
    type="[ image ]"
    top="pixels:0"          left="pixels:0"
    wid="pixels:0"
    depth="[ screen | topMost ]"
    frameCol="0xrgb"
    frameWid="pixels:0"
    geoX1="pixels"         geoX2="pixels"
    geoY1="pixels"         geoY2="pixels"
    geoLat1="latitude"     geoLat2="latitude"
    geoLon1="longitude"    geoLon2="longitude"
</resource>
```

// Resource object
// media type
// spacial origin of image
// width if non-zero
// stuck to screen / freestanding
// color of image frame, if any
// width of frame, if any
// Raster map x references
// Raster map x references
// Latitude reference points
// Longitude reference points

Perhaps the simplest kind of resource is an image file, which might contain an image to display. The following line identifies an image on the server, loads it for future use, and makes it available to be instantly shown by referring to it by the name "myPic."

```
<resource id="myPic" type="image" src="http://virginia.edu/pic.jpg" />
```

Movie / Audio / Animation Resources are Flash video formatted files (.FLV), MP3 audio files and SWF flash files. The *autoplay* tag which determines if the movie playing when it first appears. Omitting the *wid* tag will cause movie and player to size itself to match the native resolution on a Flash movie. Setting the *glue* to some glue object will cause that glue object to be called every *n* ms specified by *time*. *Start* and *end* specify the movies bounds. You can specify a movie on YouTube by pre-pending "you://" to the YouTube *v= ID* tag in the video's url (i.e. YouTube link www.youtube.com/watch?v=QaRfxjcpYvM&feature=pop would be: *src="you:// QaRfxjcpYvM"*).

| | |
|--|-------------------------------------|
| <resource> | // Resource object |
| type="[movie audio flash]" | // media type |
| src="url" | // address of media file |
| autoplay="[true false]" | // play movie when visible |
| autoRewind="[true false]" | // rewind movie after end |
| timer="ms:250" | // time between glue calls, in ms |
| glue="glueID" | // glue to run each timer interval |
| start="sec:0" | // start of movie, in seconds |
| end="sec" | // end of movie MUST BE SET! |
| <textFormat> <i>textformat</i> </textFormat> | // specific text attributes |
| <frame> <i>frame</i> </frame> | // box of timeline |
| </resource> | |

2. Data Resources

In its simplest form, a data resource is a list of things: numbers, words, paragraphs, URLs, etc. Data can be brought into projects in four ways. 1) By specifying the listing in the XML directly; 2) by accessing an XML file somewhere on the web via a URL; 3) a query to a MySQL server; or 4) using a web service.

Each method stores the data into an identical format with the HistoryBrowser, as a list containing the items, referenced from the resource's name. Data brought in this way can be queried using the GLUE query() method or accessed in a number of ways.

1. Defining data directly in the XML:

```
<resource id="myData" type="sdata" src="1,2,3,4,5" />
Will create a list of 5 items (1,2,3,4,5) within the resource accessible through
myData.data1.
```

```
<resource id="myData" type="xydata" src="10;5,20;6,30;7" />
Will create a list of 3 items within the resource accessible through myData.data1
(10,20,30) and myData.data2 (5,6,7) useful for paired x y coordinate data.
```

2. Accessing data via an XML file:

```
<resource id="myData" type="xml" src="http://mysite.org/data.xml"/>
Will load a file called data.xml on the server at mysite.org. That file can have any number
of fields and rows. The project tool has a converter that takes tab-delineated spreadsheet
files and formats it automatically. The actual format is listed in the appendix. The data is
accessed by its field's name (i.e. myData.censusAge).
```

3. Accessing data via a MySQL query:

```
<resource id="sqldata" type="mysql" host="dbm1.virginia.edu"
name="myDB" user="mst3k" password="1234"
query="SELECT * FROM mapid ORDER BY id" />
```

Will send a query to the defined MySQL database and retrieve the results in the same format as an XML formatted query (#2 above). The user and password fields are encoded for privacy.

Accessing individual data elements in a data set

Data sets are typically accessed by querying the data with a query() glue method, but you can access individual elements by specifying them. For example, if we had a resource with the id of *myData*, `status(*myData.name)` would print a list of all the rows of the *name* field on the screen and `status(*myData.name.1)` would print the 2nd name (the count starts at zero).

3. Map Resources

[TBD]

```
depth="[ screen | topMost ]" // stuck to screen / freestanding
onclick="glueID" // glue to run if feature clicked
ondblclick="glueID" // glue to run if double-clicked
onhover="glueID" // glue to run if hovered over
cols[] // list of feature interior colors
edg[] // list of feature edge colors
```

```
<resource id="name" > // Resource object
  type="[ data | map | image | graph | table | movie | audio ]" // media type
  query="sql" // actual SQL query
  src="url[/db]" // web address of resource
  store="[ live | new | local ]" * // data storage options
  xy="x[0],y[0]; x[1],y[1];... x[v],y[n]" // xy data
  preload="[ true | false ]" // hold upstart until it's loaded
  geoLat1="latitude" geoLat2="latitude" // Latitude reference points
  geoX1="pixels" geoX2="pixels" // Raster map x references
  geoLat1="longitude" geoLat2="longitude" // longitude reference points
  geoY1="pixels" geoY2="pixels" // Raster map y references
  onclick="glue" // glue to run if clicked on
</resource>
```

4. InfoBox Resources

Information boxes are popup boxes used to display textual information on demand. They are typically called by clicking on path and graph elements. InfoBoxes can contain a variant of HTML formatting and can be populated using search and replace variable that can be set using a database. The appendix contains detailed information on the text formatting options available. See the example in the **Cookbook** of how to set up an infoBox.

```
<resource id="name" > // Resource object
  Text to display
  title="Name" // short title of resource
  type="infobox" // resource type
  close="[ true | false ]" // has closing button?
  selectableText="[ true | false ]" // has selectable text?
  scroller="[ true | false ]" // has scroll bar?
```

```

border="pixels:24" // indented border around text
tail="[ line | none | solid ]" // tail to click point
position="[ abs | north | south | east | west ]" // position relative to click point
</resource>

```

The text can be have special tags(\$\$1 through \$\$99 that can be replaced dynamically using the **replaceword()** method. Text can contain the standard text formatting macros (see appendix).

For example this script:

```

<resource id="myInfoBox" type="infobox" tail="line" >This is $$1 and this is $$2 />
<glue id="fillIt" from="myInfoBox" >
  list($v,one sentence,the second)
  replaceword(myInfoBox,$v)
</glue>

```

Would result in a box with the following text: *"This is one sentence and this is the second"*

Containers

Containers are collections of objects used to put information on the screen on top of a resource, such as an image or a map. There are currently three types of a container: A **Path** creates trails and navigation objects, a **CMap** makes concept maps, a **Dock** mimics the action of the Apple Macintosh dock, and a **Canvas** defines a scrollable area to contain images. Drop-shadows can be added via frame objects. Containers place **dots** in particular places on the screen. A dot can be a graphic shape, such as a circle or square, an image, or an icon. Dots can have **Glue** methods associated with them so actions can occur when you click on them.

| | | |
|---------------------|--|-------------------------------------|
| <dot | id="name"> | // Dot marker object |
| | col="0xrgb:0x000000" | // color of dot |
| | wid="pixels:0" hgt="pixels:0 | // width / height of dot |
| | rot="degrees:0" | // angle of rotation |
| | alpha="opacity:100" | // 0-100 opacity |
| | style="[_ bar cir star triu trid tril trir rbar icon: .jpg/.gif/.png]" | // marker shape |
| | x="pixels" y="pixels" | // location for dot in pixels |
| | lat="latitude" long="longitudez | // location for dot in lat / lon |
| | date="day/mo/year" -or- time="0-1:-1" | // time 0-1 (-1=no time) or date |
| | pct="Number 0-1" | // used in route only |
| | lab="string" | // label for dot marker |
| | glue="glue" | // glue to activate if clicked |
| | hover="glue" | // glue to activate if hover'd over |
| | dropWid="pixels:0" | // width of drop shadow |
| | dropBlur=" pixels:0" | // blur of drop shadow (0-9) |
| | labelPos="[top bot center]" | // position of label |
| | <frame> frame </frame> | // frame to |
| </dot> | | |

NOTES:

1. Dots will continue using properties set in previous dots to reduce unnecessary repeating of attributes. For example, if you set the style to "triu" (up-facing triangle), all dots that follow would be rendered as "triu" until re-specified.
2. Clicking on a dot will cause a GLUE element to run if there is one specified, allowing you to trigger other actions and displays. You can find out which dot was clicked by looking at the \$\$click global list parameter, which will be set to the dot's index in the path. The first dot will set \$\$click to 0, the second to 1, etc. Alternatively, you can set this value manually by appending it the glue's name with a ? mark, such as glue=myGlue?show me. This will cause the words "show me" to be set in the \$\$param list and available for use in the glue script.

PATH

Paths place **dots** on the screen and can be connected by lines if desired. The width, color, and alpha can be specified. The position of the dots is set in pixels, relative to the base resource the path is atop, or in lat / lon coordinates based on the base resource.

If *showAllDots* is set *true* the dots are not time dependent. If *true*, dots can have times associated with them, so they will appear when the view's timeline date reaches a certain time. The time or date attribute of a dot tells when that dot will be drawn. 0 is at start, .5 is middle, 1 is end, etc. If a date is set, dot will draw when that date matches date on timeline. Time dependent paths can have the line advance between dots as the time changes by setting *tweenLines* to *true*. The current time of the line can be preceded by an icon by setting the *head* attributes. Paths are useful in showing a trail on a map, but are often used to put buttons, menus and other navigational elements on the screen. Set *showAllDots* to *true*, so they will always appear See the example in the **Cookbook** of how to set up a path or a menu.

| | |
|--|---|
| <code><path id="name" ></code> | <code>// Path object</code> |
| <code>col="0xrgb: :0x000000"</code> | <code>// color of inter-dot lines</code> |
| <code>wid="pixels:0"</code> | <code>// width of lines (0 = none)</code> |
| <code>alpha="opacity:100"</code> | <code>// 0-100 opacity</code> |
| <code>res="resourceID"</code> | <code>// Resource to pull GIS info</code> |
| <code>showAllDots="[true false]"</code> | <code>// show all dots always</code> |
| <code>tweenLines="[true false]"</code> | <code>// animate line between dots</code> |
| <code>headStyle=" [icon:name .jpg/.gif/.swf]"</code> | <code>// Leading line icon style</code> |
| <code>headSize="pixels" headCol="0xrgb"</code> | <code>// Leading line icon width/color</code> |
| <code>headEnd=" [true false]"</code> | <code>// Leave icon on when done</code> |
| <code>glue="glue"</code> | <code>// glue if head is clicked</code> |
| <code><textFormat> textformat </textFormat></code> | <code>// specific text attributes</code> |
| <code><dot> dot </dot> ...</code> | <code>// marker dots</code> |
| <code><pathway> pathway </pathway> ...</code> | <code>// pathways containing dots</code> |
| <code><route> route </route> ...</code> | <code>// draw pathways</code> |
| <code></path></code> | |

If you have a number of journeys along a set number of path ways, you can define a collection of dots as a **pathway**. The timing within the path is relative from 0 to 1 start to end, rather than a particular date for better flexibility and accomplished by setting the *pct* attribute in the dots contained in the pathway. That **pathway** can be drawn multiple occasions and different times by adding **route** elements to a **path** that define the *start* and *end* times a particular *pathway* (containing the **dots**) will be drawn. See the example in the **Cookbook** of how to set up a route.

| | |
|---|---|
| <code><pathway id="name" ></code> | <code>// Pathway object</code> |
| <code><dot> dot </dot> ...</code> | <code>// marker dots</code> |
| <code></pathway></code> | |
| <code><route></code> | <code>// Route object</code> |
| <code>pathway="ID of pathway"</code> | <code>// id of pathway to draw</code> |
| <code>start="date"</code> | <code>// route start date</code> |
| <code>end="date"</code> | <code>// route end date</code> |
| <code>glue="glue"</code> | <code>// glue if head is clicked</code> |
| <code></route></code> | |

Clicking on a head will cause a GLUE element to run if there is one specified, allowing you to trigger other actions and displays. You can find out which head was clicked by looking at the `$$click` global list parameter, which will be set to the dot's index in the path. The first head in the first path will set `$$click` to 1000, the second to 1001, etc. Alternatively, you can set this value manually by appending it the glue's name with a ? mark, such as `glue=myGlue?show me`. This will cause the words "show me" to be set in the `$$param` list and available for use in the glue script.

CONCEPT MAP

Concept maps are similar to paths, but the paths can be arranged in a radial manner similar to a hub and spoke shape. The dots are not time dependent, and lines (edges) must be specifically drawn by setting the relationships between the dots (nodes). Labels are automatically drawn if specified underneath the dot. The frame specifies the overall bounds of the concept map.

You can add a **legend** that identifies the type of lines by including a legend tag. Each tag adds an entry that shows a label associated with each **linestyle**. Setting backCol will draw a “wash” of color alpha'd over the background, to help highlight the concept map.

```
< cmap id="name" >                                     // Concept map object
  shape= "[ radial ]"                                   // overall text attributes
  <textFormat> textformat </textFormat>                // box of callout
  <frame> frame </frame>                                // stagger length of "spokes"
  stagger="pixels:0"                                    // background "wash"
  backCol=0xrgb                                        // nodes
  <dot> dot </dot> ...                                  // edges
  <line> line </line> ...                               // edge types
  <lineType> lineType </lineType> ...                  // legend entries
  <legend> legend </legend> ...
</ cmap >
```

The **lines** define the relationship between the dots and determine how they will be placed. Setting the *from* tag to "" will position the dot pointed by the *to* tag it at the center of the concept map. The **linestyle** object defines how the lines will be drawn, and the letter that will be drawn in the middle. See the example in the *XML Cookbook* of how to set up a radial concept map.

Dots are typically arranged automatically, but you can arbitrarily place a dot anywhere on the screen by setting the *x* and *y* **dot** attributes to a position and setting the **line's** *dir* attribute to float. If you have specified a line style, the line will be drawn from the center of dot specified in the **line's** *from* attribute to the center of the dot.

```
< line >                                               // Connector line (edge) object
  from="dotID"                                         // node connected from
  to="dotID"                                           // node connected to
  style="lineStyleID"                                  // type of connection
  dir="[ one | two | float ]"                          // direction
</ line >

< legend >                                             // Legend
  style="lineID"                                       // id of line style
  lab="String"                                         // text to display
</ legend >

<LineStyle>                                           // Connector lineType object
  col="0xrgb"                                          // color of dot maker
  wid="pixels:0"                                       // width of dot marker
  alpha="opacity:100"                                  // 0-100 opacity
  letter="letter"                                       // width of dot marker
  lab="label"                                          // rollover text
  type="[ isa | partof | contains ]"                  // type of connection (TBD)
  arrows="[ from | to | both | none ]"                // line ends (TBD)
</LineStyle>
```

DOCK

A **dock** object presents a series of dots horizontally across the screen in a similar fashion to the application dock used in the Apple Macintosh OSX. The dots are typically icons or images that are fixed to a base bar. As the mouse hovers over one, it and its neighbors grow by the percentage spec'd by the *growth* tag. Setting the *growStyle* to "single" will cause only the dot being hovered on to grow while hovered over, as opposed to the default of "taper", which also grows the two dots on either side of the one being hovered over as well. The dots can have glue attached to cause some action when clicked.

The *frame* object sets the bounds of the dock, but since the dock grows and shrinks based on the number of *dots* within it, the dock will draw from the center of area defined by the frame's *left* and *wid* tags. The frame's *hgt* tag defines the height of the base bar. Setting the *hgt* to 0 will inhibit the drawing of the base bar.

```
<dock id="name" >                                // Dock object
  <textFormat> textformat </textFormat>           // specific text attributes
  <frame> frame </frame>                           // box of base bar
  growth="Percentage:200"                          // size to expand when hovered
  growStyle="[ single | taper ]"                   // style of growth when hovered
  <dot> dot </dot> ...                             // dots in container
</dock>
```

DOCVIEWER

An **docviewer** is resource very similar an *infobox* to that can hold HTML formatted text and a picture side-by-side in series of pages provided by a data source (i.e. and XML file or SQL query). The data source can have 4 fields: *title*, *source*, *desc* and *caption*. The *title* field provides a title at the top and a way to select items from the data source. Items with the same title will appear as pages within the document viewer. The *source* field gives a url for a picture if desired, and *desc* is an html formatted text area. If a *caption* field is defined, it will appear underneath the picture.

If both *desc* and *source* are defined, they will appear side by side. If only one is defined, only that one will appear. The text and picture information is supplied by the *filldocviewer* glue method, typically as the result of a query method. Text can contain the standard HTML formatting macros (see appendix).

See the section in the appendix on "Making Booklets" for more information on creating the contents of docviewers/

```
<resource id="name">                               // docviewer object
  selectable="[ true | false ]"                   // text can be selected by mouse
  scroller="[ true | false ]"                     // has a scroller bar
  type="docviewer"                                // set as docviewer
  <textFormat> textformat </textFormat>           // overall text attributes
  <frame> frame </frame>                           // box of callout
</resource>
```

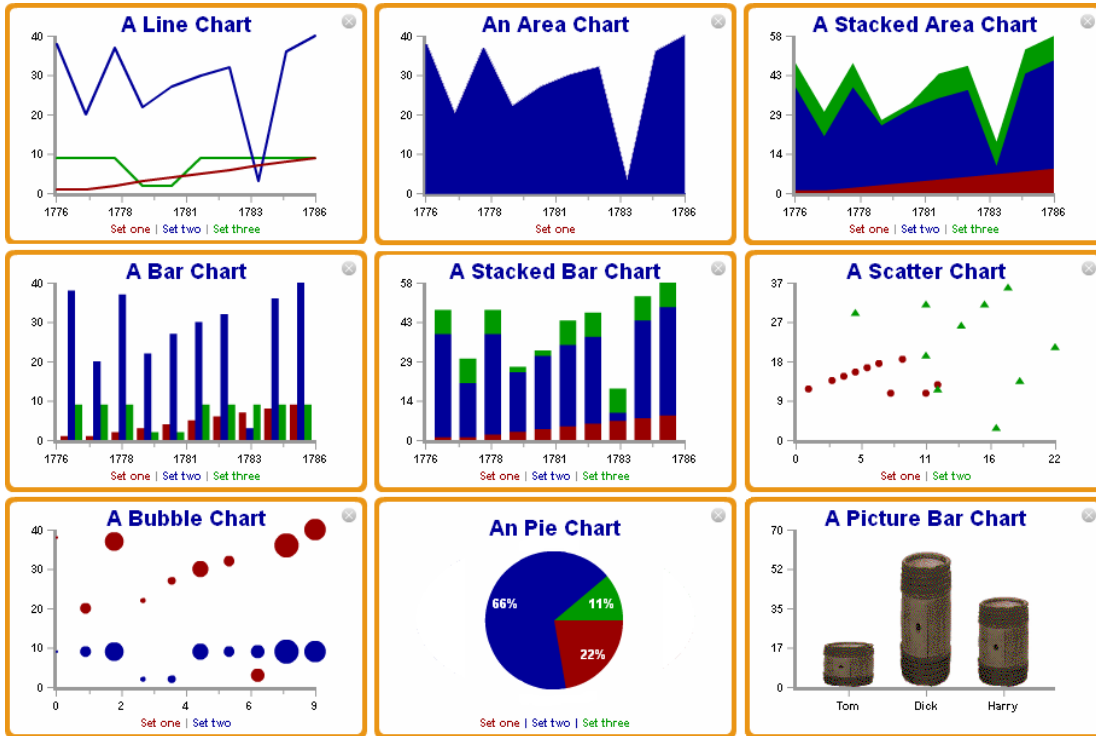
For example this script:

```
<resource id="myData" type="xml" src="myDataFile.xml" />
<resource id="myDocView" type="docviewer" />
<glue id="fillIt" from="myDocView " >
  filldocviewer (myDocView,Overview,myData)
</glue>
```

Would load an XML file called *myDataFile.xml* , when *fillIt* was called the docviewer would be filled by any pages in the xml file with the title of "Overview."

Data Graphing

The HistoryBrowser supports a number of chart types that can be drawn, including line, area, stacked area, bar, stacked bar, scatter, bubble, picture, and pie charts as shown here:



The charts can have multiple data sets, the color and labels of each are defined by the **marker** item. Charts can have x and y axis by adding an **xAxis** or **yAxis** item to the resource definition, The *title* and *subtitle* attributes allow you to add titles and subtitles to the graph. The bar and area charts can have their data sets stacked by setting the *stacked* attribute to true. Setting the *legend* attribute to true will show whatever **marker names** were set for that dataset at the bottom.

Scatter and bubble charts are bi-variate, requiring 2 datasets for each plotted set. On scatter charts, the first sets the position on the X-axis and the second one sets the position along the Y-axis. On bubble charts, the dots are plotted along the X-axis like a line chart, but the first data set controls the size of the dot drawn at each point. The bubbles are scaled according their value relative to the largest data value in that first set. The dataset's **marker wid** attribute sets the maximum size of the bubbles when the data value is the highest.

Pie charts get their label names and colors from the **marker** tags. There should be one **marker** for each pie slice. You can have the slice values printing inside each slice by setting the *showValues* attribute to "true", or "percent" if you want the slice's percentage to the whole.

Use the **dataset()** GLUE method to set the data sets with values. You can easily animate charts by using the **tweenlist()** GLUE method to transition between two lists of data.

The charts pictured above were made using slight variations of the following script on the following page:

```
<resource id="myGraph" type="graph" style="area" stacked="false" border="35"
  title="A Chart" legend="true">
  <textFormat col="0x000099" size="18" bold="true"/>
  <frame wid="300" hgt="200" left="24" top="160" corner="8" frameWid="4"/>
  <xAxis col="0x999999" majorTick="8" minorTick="6" wid="3"
    min="1776" max="1786" showValues="true" lab="one|two|three"/>
  <yAxis col="0x999999" majorTick="8" wid="3" min="0" max="80" mod="1"
    showValues="true"/>
  <marker col="0x990000"/>
  <marker col="0x000099"/>
  <marker col="0x009900"/>
</resource>
<glue from="myGraph" init="true">
  list($myData1,1,1,2,3,4,5,6,7,8,9)
  list($myData2,38,20,37,22,27,30,32,3,36,40)
  list($myData3,9,9,9,2,2,9,9,9,9)
  dataset(myGraph,0,Set one,$myData1)
  dataset(myGraph,1,Set two,$myData2)
  dataset(myGraph,2,Set three,$myData3)
</glue>
```

| | |
|---|-------------------------------|
| <resource id="name"> | // Graph object |
| close="[true false]" | // show close button |
| type="graph" | // set as graph |
| glue="glueID" | // glue if clicked |
| title="name" | // title of graph |
| showValues="[true percent none]" | // show data values in pie |
| subtitle="name" | // subtitle of x axis |
| highWid="pixels" | // width of highlight bar |
| type="[bar line area scatter]" | // type of graph |
| border="pixels" | // space around data area |
| stacked="[true false]" | // are data elements stacked? |
| <legend show="[true false]" /> | // show legend |
| <textFormat> textformat </textFormat> | // specific text attributes |
| <marker num="number" marker="marker" title="name" />... | // data set info |
| <xAxis> axis </xaxis> | // X axis |
| <yAxis> axis </yaxis> | // Y axis |
| </resource> | |

| | |
|--|-------------------------------------|
| <marker id="name"> | // Marker object |
| type="[bar box triangle dot line image.jpg]" | // type of marker |
| col="0xrgb" | // color |
| wid="pixels:10" | // width of marker |
| datawid="number:2" | // width of data (i.e. line or bar) |
| name="name" | // name of marker |
| <textFormat> textformat </textFormat> | // specific text attributes |
| </marker> | |

| | |
|--------------------------------------|-------------------------|
| <xAxis> <yAxis> | // Axis objects |
| title="name" | // axis title |
| col="0xrgb" wid="pixels" | // color and line width |

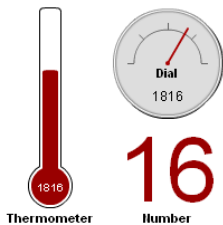
```

majorTick="pixels"      minorTick="pixels"           // major/minor tick lengths
grid="[ true | false ]" // show gridlines
valueCol="0xrgb"        // color of values, if any
lab="a|b|c"             // x labels separated by |'s
min="number" max="number" mod="number"           // data range and mod
showValues="[ true | false ]" // show data values
pos="[left | right]"   // position (yaxis only)
<textFormat> textformat </textFormat>          // specific text attributes
</xAxis>

```

Widgets

Widgets are a type of graph that graphically displays a single continuous value on the screen, such as a dial, clock, thermometer, etc. The range of widgets available will grow with time, but they plot the *val* attribute from *min* to *max*.



The data is plotted in the color *col*. The title is displayed below the widget except for the dial, where it's in the dial. The value is displayed to 2 decimal places if it is less than 1, or otherwise whole numbers. The size of round widgets like dials look at the *wid* attribute, where things like thermometer use the *hgt* attribute as well.

You can set the *val* in glue methods, like this `set (*myWidget.datVal, 25)`, which would set the value of the widget with the id of "myWidget" to 25. See the example in the cookbook.

```

<widget id="name">
  title="name"
  col="0xrgb:0x990000"
  styl="clock[dial|number|thermometer]"
  wid="pixels" hgt="pixels" top="pixels" left="pixels"
  min="number:0"      max="number:100"
  val="number:50"
  <textFormat> textformat </textFormat>
</widget>

```

// Widget object
// title
// color of inter-dot lines
// type of widget
// size and position
// data range
// Initial value
// specific text attributes

Icon Types

There are a number of icons that can be used on dots and markers. By adding *icon:* to the name (i.e. *icon:comment*) they can be used as vector-based artwork. They can be scaled up and down by setting the *wid=""* tag to the desired width, and rotated any angle using the *rot=""* tag. If you don't specify a size, the width listed will be used

| Name | Description | Width |
|------------------|--|--------------|
| blackbar | A filled black bar | 10 |
| blackbox | A hollow black box | 10 |
| circle | A white circle with grey drop-shadow | 95 |
| comment | A white circle with "tail" and grey drop-shadow | 86 |
| document | A white box with page lines and "dog-eared" corner | 20 |
| dottedbox | A hollow black box drawn with dot | 20 |
| info | A white circle with "I" and black border | 30 |
| letter | A white envelope with black lines and grey drop-shadow | 54 |
| moviecam | A grey drawing of a movie camera | 53 |
| person | A white circle with grey drop-shadow and grey silhouette | 95 |
| radialmap | Hub and spokes in white circle | 50 |
| slate | Black and white movie slate | 30 |
| tree | Architectural Green tree (plan view) | 20 |

Dot Style Types

There are a number of drawn shapes that can be used on dots and markers. They can be scaled up and down by setting the *wid=""* tag to the desired width. Setting the *col=""* tag will set the color they will be drawn in.

| Name | Description |
|-------------|-----------------------------------|
| bar | A filled bar |
| cir | A filled circle |
| rbar | A filled bar with rounded corners |
| star | A filled 5-point star |
| trid | A filled triangle facing down |
| tril | A filled triangle facing left |
| trir | A filled triangle facing right |
| triu | A filled triangle facing up |

Common Objects

| | |
|--|---|
| <pre><textFormat id="name"> color="0xrgb:0" alpha="opacity:100" size="pixels:12" bold="[true false]" italics= [true false]" underline= [true false]" face="[_sans _serif _fixed specificFont]" slant="degrees:0" leading="pixels:auto" align="[left right center justify]" </textFormat></pre> | <pre>// Defines text formatting // text color // 0-100 opacity // size // bold // italics // underline // font face // orientation in degrees // total height between lines // horizontal alignment</pre> |
| <pre><frame id="name"> id="name" title="name" wid="pixels" hgt="pixels" top="pixels" left="pixels" corner="pixels:0" alpha="opacity:100" docking="[left right top bottom center float]" backCol="0xrgb:0xfffff" frameCol="0xrgb:0x000000" frameWid="pixels:0" dropWid="pixels:0" dropBlur=" pixels:0" </frame></pre> | <pre>// Holds visual items // frame's id // title of frame // size and position // for rounded rectangles // 0-100 opacity // docking mode // interior color // color of frame // width of frame // width of drop shadow // blur of drop shadow (0-9)</pre> |
| <pre><shapedata> col="0xrgb" edgeCol="0xrgb" edgeWid="pixels" xOff="pixels:0" xOff="pixels:0" <polygon polyline arrow text id="name" xy="xydata" col="0xrgb" edgeCol="0xrgb" edgeWid="pixels /> </shapedata></pre> | <pre>// Shape object // default interior color // default color of edge // default edge wid (0=none) // offset of image to screen // element id // name of element // cords (x,y; ... x,y;) // color info</pre> |

GLUE

Glue is used to connect the data resources described above to data consumers such as display tables, popup windows, charts, and data-driven maps.

Glue is written as a series of lines of text in the XML file that drives the HistoryBrowser project. A view may contain any number of lines of Glue and each line has an *id*, or name, so you can string together multiple lines, or cause a line to execute from an item in an control panel.

Glue also provides an opportunity to calculate tables and fields in resources based on a simple script in the tag. Many common types of operation can be defined between these elements, so that the HistoryBrowser is able to relate rich data relationships between them and visualize them on a special and temporal basis.

```
<glue id="name"                               // GLUE object
      init="[ true | false ]"                 // run glue at startup?
      once="[ true | false ]"                 // run glue only once?
      calculation script                       // calculation script (optional)
</glue>
```

The *from* attribute specifies what resource to display *to* attributes says where to place the results. If only a calculation between data elements is wanted, *from* can be empty. Setting the *init* attribute to true will cause the glue element to be executed at startup. Setting the *once* attribute to true will cause the glue element to be executed only once. This is useful for setting data sources, filling routes, etc.

List Variables

HistoryBrowser scripts support two kinds of list variables. These lists can contain one or more items. All lists names are preceded by a dollar sign (i.e. \$myList). A normal list is only active (has scope) within the GLUE method it is being used. Global lists (meaning their scope is through-out the current view) are preceded by 2 dollar signs (i.e. \$\$myGlobalList).

You can get various elements of a resource, such as an xml data source by referencing the resource's name with a star and the specific field you want to get. For example, if we had an xml resource called "myData" with a field called "name" in it, `status (*myData.name)` would show all the names in the file, and `status (*myData.name.3)` would show the 4th name in the xml file (remember that computers start counting at zero).

There are a number of global list variables that are useful to see what time the timeline is at and what dot or map feature was clicked on:

| | |
|---------------------|---|
| \$\$click | Gives the index of the currently clicked on map feature or dot |
| \$\$param | If you had specified a parameter by adding "?xxxx" to the glue spec, xxxx will be passed to the glue method |
| \$\$now | The time in the timeline from 0-1 |
| \$\$curYear | The current year in the timeline |
| \$\$curMonth | The current month in the timeline as mo/year |
| \$\$curDays | The current time in the timeline as days +/- 1970 |

Comments

You can comment out lines of glue script by using `/* */` to bracket the area, like this:

```
list($myData1,1,1,2,3,4,5,6,7,8,9)
list($myData2,38,20,37,22,27,30,32,3,36,40)
/* list($myData3,9,9,9,2,2,9,9,9,9,9)
dataset(myGraph,1,Set two,$myData2) */
dataset(myGraph,2,Set three,$myData3)
```

Or use `//` to comment from that point to the end of the line. This is also useful for documenting the script's functions:

```
list($myData1,1,1,2,3,4,5,6,7,8,9) // Data set 1
list($myData2,38,20,37,22,27,30,32,3,36,40) // Data set 2
//list($myData3,9,9,9,2,2,9,9,9,9,9) // Commented out
```


COPY

This method will copy a member or members from one resource or list to another. If the `destID` is prefaced with “\$\$”, a global list will be created if it doesn’t already exist, whose scope is beyond the current calculation script.

copy(destID, destStart)

`destID:String`
`sourceID:String`

Name of list to or resource to copy to
Name of list to or resource to copy from

SELECT

This method selects one member of a *source* list based on the first member of a *which* list and places it in the *destination* list.

select(source, destination, which)

`source:String`
`destination:String`
`which:Number`

ID of list of values to select from
ID of list where selection is placed
ID or number of selection number

SEGMENT

This method will sort data into a number of preset categories and use those as criteria to create a new list.

segment(sourceID, destID, filters, values)

`sourceID:String`
`destID:String`
`filters:String`
`values:String`

ID of source data resource
ID of destination data resource
ID of list of numbers to segment data
ID of list of values to assign segmented data

As an example, suppose we wanted to color a map so that populations of different area are drawn in different colors. Areas with no people should be colored white, populations from 0-25 colored light red, 25-50 medium red, 50-75 red, and population greater than 75 colored bright red:

```
<resource id="myData" type="data" src="http://mysite.com/pop1845.xml" />
<resource id="myMap" type="map" src="http://mysite.com/myMap.xml" />
<glue from="myMap" init="true">
  list($slots,0,25,50,75)
  list($colors,0xffffffff,0x330000,0x990000,0xff0000)
  segment(myData.pop, myMap.col, $slots, $colors)
</glue>
```

The segment method can be used to figure out which slot a time period figures in. Whatever date the timeline (the property called *now*) is at is compared to the slots, and the `mapNum` list is set to number from 1-the number of slots. (i.e. 1855 = 1, 1860=2, 1890=4)

```
list($slots,1850,1860,1870,1880)
list($mapNum,0)
segment($$now, $mapNum, $slots, null)
```

MOVE

This method will move a resource over time. If the timing is set to 0, the resource will always be positioned at the starting positions specified. An id of *screen* can be use to move entire screen.

| | |
|--|---|
| move(resourceID, startX, startY, startZ, endX, endY, endZ, timing, eases) | |
| resourceID:String | ID of resource or <i>screen</i> |
| startX:Number | starting horizontal position |
| startY:Number | starting vertical position |
| startZ:Number | starting zoom position |
| endX:Number | ending horizontal position |
| endY:Number | ending vertical position |
| endZ:Number | ending zoom position |
| timing:String | ID of timing source (i.e. timeline, var, 0) |
| eases:Number | motion slows (0=none1=start 2=end 3=both) |

TWEEN

This method will set a resource field to some position over time. If the timing is set to 0, the resource will always be positioned at the starting positions specified.

| | |
|--|---|
| tween(fieldID, start, end, timing, eases) | |
| fieldID:String | ID of field, with '.' modifiers |
| start:Number | starting value |
| end:Number | ending value |
| timing:String | ID of timing source (i.e. timeline, var, 0) |
| eases:Number | motion slows (0=none1=start 2=end 3=both) |

LINK

This method will cause a webpage to open. The "http://" portion of the URL is not required. You can specify the name of a *list* method in place of a URL, in which case, the URL name can respond to a click, say from a path object. Target sets where the page will open, which can be set to the frame's name or the preset values of *_blank*, *_self*, *_parent*, or *_top*. The *clickParam* will cause the current click parameter (0 if none) to be appended to the url as *?id=#* (or *&id=#* if there is a name=value pair already there.)

| | |
|--------------------------------------|--|
| link(url, target, clickParam) | |
| url:String | full URL of page to load, or ID name of list |
| target:String | browser window or frame |
| clickParam:Boolean | if set to <i>true</i> , <i>?id=</i> will be added to url |

***About the *clickParam* value:**

When a map is clicked on, the feature number associated with the feature clicked on will be available to methods that support the *clickParam* option, such as the **link** method.

SHOW

This method sets the visibility of a resource. The resource can be rendered fully transparent (*opacity=0*) to fully opaque (*opacity=100*) or any point in between.

show(resourceID, opacity)

resourceID:String
opacity:Number

ID of source data resource
Opacity of resource 0-100

REFRESH

This method will cause the resource identified to be redrawn.

status(resourceID)

resourceID:String

ID of source data resource

DISSOLVE

This method will dissolve between two resources. Times are expressed as 0-1, with one being the length of the timeline and 0 its start.

dissolve(inID, outID, start, mid, end, dur)

inID:String
outID:String
start:Number;
mid:Number;
end:Number;
dur:Number

ID of incoming resource
ID of outgoing resource
Start time of outgoing res (0-1)
Start time of incoming res (0-1)
End time of incoming res (0-1)
Duration of dissolve transition (0-1)

RADIOSHOW

This method acts like a radio button, and sets the visibility of a list of resources such that only one is visible at any given time. The selected resource can be rendered fully transparent (*opacity=0*) to fully opaque (*opacity=100*) or any point in between. All others are hidden. Setting *select* to 0 hides them all. The *select* can also reference an ID of a list. When using *radioshow* to *select* between dot object, use the word "dot" as the *resources* list.

radioshow(select, opacity, resources)

select:Number
opacity:Number
resources:String

which resource 1-N
Opacity of selected resource 0-100
ID of list of resource IDs

STATUS

This method prints a message in the status area at the bottom of the screen.

status(message)

message:String

ID of list with message, or literal

DATASET

This method adds a row of data to a graph.

dataset(graphID, set, legend, data)

| | |
|----------------|-----------------------------|
| graphID:String | ID of source graph resource |
| set:String | number of data set |
| legend:String | legeng |
| dataID:String | ID of data to add |

DATETIME

This method sets the amount of data to show in a graph to make time series data appear with, such as following the position of a timeline. (use the `$$now` parameter to track timeline from 0-1). On line and area charts, the number of points shown on the x-axis will be mediated by the time set. For example, if a chart has 30 elements, setting time to .1 will show the first 3.

datetime(graphID, time)

| | |
|----------------|------------------------------|
| graphID:String | ID of source graph resource |
| time:Number | Amount of data to show (0-1) |

NORMALIZEGRAPH

This method will set the status of a graph set by *graphID* to plot the data as raw numbers by setting *max* to 0 (it's default condition) or normalize the data from 0 to the number set by *max*, typically 100. This is useful when trying to compare datasets with wildly different ranges.

normalizegraph(graphID, max)

| | |
|----------------|-----------------------------|
| graphID:String | ID of source graph resource |
| max:Number | Maximum number on Y axis |

REPLACEWORD

This method looks at some text and replaces special symbols with a word or words. The symbols such as `$$1`, `$$2`, etc., where the `$$` identifies it as a symbol and the number following it says which one in the list it should be replaced with. The replacement parameter is the ID of a list of replacement word or words. `$$1` would be replaced by the first member in the list, `$$2` would replace the second member, etc.

replaceword(textID, replacements)

| | |
|---------------------|--|
| textID:String | ID of resource containing text with symbols |
| replacements:String | ID of list of values to replace symbols with |

```
<resource id="myBox" type="infobox" position="north">
  <![CDATA[
    This is the $$1, this is the $$2, and this is the $$3.
  ]]>
  <frame wid="200" hgt="150" " backCol="0xffffcc" />
</resource>
```

```

<glue from="myBox" init="true">
  list($words,first,second,third)
  replaceword(myBox,$words)
</glue>

```

Will result in: *This is the first, this is the second, and this is the third.*

FILLDOCVIEWER

This method will fill a document viewer object with data from a data source (i.e. A an XML file, or a SQL database query). You can select a specific item in the data source by setting the title parameter in the glue call to the item's number prefaced with a # sign (i.e. #32).

filldocviewer(viewerID, title, dataID)

| | |
|-----------------|---------------------------|
| viewerID:String | ID of docviewer resource |
| title:String | Title to look for in data |
| dataID:String | ID of data resource |

DOTFILL

This method will fill a container object, such as a path or concept with dot data from a data source (i.e. A an XML file, or a SQL database query). The data source must contain at least the x,y, and time fields. See the dot specification for more information.

dotfill(containerID, dataID)

| | |
|--------------------|-------------------------------|
| containerID:String | ID of container to house dots |
| dataID:String | ID of data resource |

ROUTEFILL

This method will fill a container object, such as a path or concept with route data from a data source (i.e. A an XML file, or a SQL database query). The data source must contain at least the start, end, and pathway fields. See the route specification for more information.

routefill(containerID, dataID)

| | |
|--------------------|---------------------------------|
| containerID:String | ID of container to house routes |
| dataID:String | ID of data resource |

IF

This method will execute the number of lines specified if condition between var1 and var2 is met.

if(var1, condition, var2,numLines)

| | |
|------------------|--|
| var1:String | Name of list, literal or resource member |
| condition:String | Condition (GT, LT, EQ NE, LE GE,LK,NL) |
| var2:String | Name of list, literal or resource member |

ADD

This method adds numbers in num1ID and num2ID and places the result in the list called destID (i.e destID=num1ID+num2ID).

add(dest, num1ID, num2ID)

destID:String
num1ID:String
num2D:String

Name of list to copy to
Name of list, literal or resource member
Name of list, literal or resource member

SUB

This method subtracts num1ID from num2ID and places the result in the list called destID (i.e destID=num1ID-um2ID).

sub(dest, num1ID, num2ID)

destID:String
num1ID:String
num2D:String

Name of list to copy to
Name of list, literal or resource member
Name of list, literal or resource member

MUL

This method multiplies num1ID by num2ID and places the result in the list called destID (i.e destID=num1ID*um2ID).

mul(dest, num1ID, num2ID)

destID:String
num1ID:String
num2D:String

Name of list to copy to
Name of list, literal or resource member
Name of list, literal or resource member

DIV

This method divides num1ID by num2ID and places the result in the list called destID (i.e destID=num1ID/num2ID).

div(dest, num1ID, num2ID)

destID:String
num1ID:String
num2D:String

Name of list to copy to
Name of list, literal or resource member
Name of list, literal or resource member

FLOOR / ROUND

The floor method returns the integral portion of num2ID and places the result in the list called destID. The round method returns the rounded value of num2ID and places the result in the list called destID.

floor(dest, num1ID) round(dest, num1ID)

destID:String
num1ID:String

Name of list to copy to
Name of list, literal or resource member

SET

This method copies srcID and places the result in the list or resource called destID.

set(destID, srcID)

destID:String

Name of list or resource member

srcID:String

Name of list, literal or resource member

LISTFILL

This method sets any values in a list called destID whose index appears in a list called srcID to the value specified in matchVal. All those not specifically in srcID would be set to the default val.

listfill(destID, srcID, matchVal, defaultVal)

destID:String

Name of list to fill

srcID:String

Name of list to specify (Null == all)

matchVal:String

Value to set matching indices in destID to

defaultVal:String

Value to set all other indices in destID to

```
list($mainList,a,b,c,d,e,f,g)
list($index,1,4,6)
listfill($mainList,$index,yes,no)
replaceword(myBox,words)
</glue>
```

Will result in a \$mainList of this:
no,yes,no,no,yes,no,yes

LISTMERGE

This method will join all the entries in a list into one entry, with a spacer between each if set. destID will create a new list, if that list does not already exist.

listmerge(destID, srcID, spacer)

destID:String

Name of list to put combined entries

srcID:String

Name of list to join into one list entry

spacer:String

Value to between entries

DATETODAYS

This method will convert a date expressed as a year, month/year, or day/month year (separators can be \ - : / or ;) into a single number representing the number of days +/- of January 1, 1970. For example, 1/1/1980 would convert to 3650 and 1/1/1960 would be -3650.

datetodays(daysID, dateID)

daysID:String

Name of list to put days into

dateID:String

Date to convert

DAYSTODATE

This method will convert the number of days +/- of January 1, 1970 into a readable date in the the form described by *format* (dy/mo/yr, mo/yr, yr, mo/dy/yr).

daystodate (dateID, daysID, format)

| | |
|---------------|-------------------------------|
| dateID:String | Name of list to put date into |
| daysID:String | Days to convert |
| format:String | Format for date |

GOTOTIME

This method will cause the timeline to go to the date specified in *when*, the number of days +/- of January 1, 1970.

gototime (when)

| | |
|-------------|----------------------------|
| when:Number | When to go on the timeline |
|-------------|----------------------------|

CALL

This method will run another glue item in the current view as a subroutine. Any parameters passed will be available in the \$\$param and \$\$click global lists.

call(glueID, params)

| | |
|---------------|------------------------------|
| destID:String | Name of glue item to execute |
| param:String | Parameters to pass to glue |

MOVIE

This method will control a movie resource's transport functions such as play or stop.

movie(resID, command, param)

| | |
|----------------|--------------------------------|
| resID:String | Name movie resource to control |
| command:String | Operation to do |
| param:String | Parameters to pass to command |

Current movie commands are:

| | |
|--------------|---|
| play | The param is the time in seconds to start playing the movie from |
| stop | The param is set to 0 |
| seek | The param is set to the time in seconds to cue the movie to |
| time | The param the name of the list to store the current time in seconds |
| start | The param is the time in seconds of movie's start time |
| end | The param is the time in seconds of movie's end time |
| load | The param is the src/path of the movie to load |

SETVIEW

Views can also be invisible and not associated with any particular tab. By setting the *visible* attribute to "true" and giving it an *id*, you can use GLUE to cause a view to show within the currently active tab's

screen space. If the view is a visible one, the view's tab will be activated. See the **setview** GLUE element and the section on the cookbook section for more information on this very powerful option.

setview(viewID)

viewID:String

Name of view to show

PLAY

play(startTime)

startTime:String

Time to start playing

This method will cause the timeline to play from the time specified in startTime. It is the same as if you dragged the timeline slider with the mouse and clicked the play button.

TWEENLIST

This method will set a list to tween (animate between to values) between two other lists over time. Useful when animating values of charts and graphs

tweenlist(destList, fromList, toList, percent, eases)

destList:String

ID of tweened list

fromList:Number

ID of from values list

toList:Number

ID of to values list

percent:Number

Percent of tween from 0-1

eases:Number

motion slows (0=none 1=start 2=end 3=both)

MENUITEM

This method will change an item in a control panel to a new title, glue or value. Leaving a parameter blank keeps the old value of it intact.

menuitem(controlID, title, glue, value)

controlID:String

ID of control panel item

title:String

Name of new title

glue:String

Name of new glue

value:String

Value of new item

APPENDIX

FORMATTING INFOBOXs and BOOKLETs

The text displayed in the InfoBox and Document Reader can be easily controlled using a subset of HTML tags shown below. Tables can be made by using <tab>'s to line up the. In the text you can use the following mark-up tags to control how the text will look:

- **b(text)** will bold the text within the parentheses
- **i(text)** will italicize the text within the parentheses
- **u(text)** will underline the text within the parentheses
- **font(face,size,color)** will size and color the text that follows
- **sp(leading,indent,tabstops)** will set the leading, indent & tabstops for text that follows
- **t()** will add a tab
- **br()** will add a line break
- **link(url,text)** will add a link to url when the text in the parentheses is clicked
- **align(side)** will set the alignment for the text that follows. Can be *left*, *right* or *center*.

Whenever possible, use the macros above to format your text, but you can also use the following tags

TAGS

| | |
|-------------------|---|
| Anchor: | |
| Bold: | |
| Font: | < font [color="#xxxxxx"] [face="Type Face"] [size="Type Size"]> |
| Italic: | <i> |
| Paragraph: | <p [align="left" "right" "center"]> |
| Underline: | <u> |
| Break: | |
| Image: | ▪ blockindent Specifies the block indentation in points.▪ indent Specifies the indentation from the left margin to the first.▪ leading Specifies the amount of leading (vertical space) between lines.▪ leftmargin Specifies the left margin of the paragraph, in points.▪ rightmargin Specifies the right margin of the paragraph, in points.▪ tabstops Specifies custom tab stops as an array of non-negative integers. |

SPECIAL CHARACTERS

| | |
|-------------------|------------------------------|
| &lt; | < (less than) |
| &gt; | > (greater than) |
| &amp; | & (ampersand) |
| &quot; | " (double quotes) |
| &apos; | ' (apostrophe, single quote) |

MACRO EXPANSION

| | |
|-------------------------------|--|
| link(url,title) | Expands to <u>title</u> |
| b(text) | Expands to text |
| i(text) | Expands to <i>text</i> |
| u(text) | Expands to <u>text</u> |
| br() | Expands to |
| t() | Expands to <tab> |
| font(face, size,color) | Expands to |
| sp(lead, indent ,tabs) | Expands to <textformat leading="lead" indent="indent" tabstops="tabs" /> |
| align(side) | Expands to <p align="side" /> |

HistoryBrowser XML Cookbook

The following topics will help you add specific elements common to many HistoryBrowser projects. Be sure to check the specific documentation described earlier for the elements involved. Using the GLUE Editor (www.historybrowser.org/edit.htm) will help you get started by providing the basic elements, automatic XML formatting, and embedded help.

1. How to set up a basic project
2. How to add a base map
3. How to add a zoom control
4. How to add a timeline
5. How create an XML Data File
6. How load an XML Data File
7. How to create an animated path
8. Adding Dots via XML to Paths
9. Using routes in animated paths
10. Adding Routes via XML to Paths
11. How to make booklets for a docviewer
12. How to create a menu using a *path* container
13. Making PopUp Information Boxes
14. How to Query Data
15. How to make a Control Panel
16. How to Make Graphs
17. How to Make Concept Maps
18. Adding Maps and Vector Graphics
19. Making a Dock
20. Using Files from Flickr
21. Adding Movies
22. Adding YouTube Movies
23. Changing a View via GLUE
24. Floating Concept Map Dots
25. Adding Widgets

1. How to set up a basic project

The following example is the bare minimum required for a HistoryBrowser project, and causes it to be shown on the screen. If you are using the GLUE Editor, it will be provided automatically for new projects when you click on the "New" menu option.

```
<project title="My Project">
  <frame wid="800" hgt="500" frameWid="1" />
  <textFormat size="11" font="_sans" />
  <tab hgt="15" wid="150" />
  <view title="My View" />
</project>
```

The first line creates an element for the entire project. The *title* attribute is internal and not shown in the project. The second line is a **frame** element that defines the basic shape for the views in the project. A **textFormat** element can also be added to control how the text will appear. See the textFormat documentation for the various options available. All other elements "inherit" this format and use it as the basis their text elements will be drawn. A **tab** element sets the size and look for the tabs that appear above the various **view** elements, where the content is shown.

2. How to add a base map

The following example adds a jpeg file to the project, and causes it to be shown on the screen. This must be added within whatever view you want the image to appear.

```
<resource id="baseMap" src=http://mySite.org/myPic.jpg />
<glue from="baseMap" init="true" />
```

The first line creates a new resource called *baseMap* from the web-accessible jpeg file called *myPic.jpg* on the *mySite.org* website. Just because you have created a resource, you wont be able to see it until it is glued to the screen. The second line causes the resource named *baseMap* to be visible every time the panel is refreshed by setting *init* to *true*.

3. How to add a zoom control

The following example adds a zoom controller to the project. This controller will zoom in and out the entire screen, in a similar way that the controller on Google maps operates. This must be added within whatever view you want the zoom control to appear

```
<zoomControl top="24" left="30" hgt="80" max="5" />
```

The *top*, *left*, and *hgt* attributes set it's position and size on the screen, in terms of pixels., The *max* attributes how many times you will be zoomed in when the scroller control is at the top (i.e. 5 = 500%).

4. How to add a timeline

The following example adds a timeline controller to the project. This controller will allow you to set a time that other elements can work with, as well as can add a player button to play animated progressions through time. This must be added within whatever view you want the timeline to appear

```
<timeline min="4/1954" max="1994" play="true" >
  <textFormat col="0x999999" size="11" font="sans" />
  <frame wid="550" hgt="550" left="120" top="530" backCol="0x999999" />
</timeline>
```

The timeline element contains a **frame** element, its *top*, *left*, *wid*, and *hgt* attributes set it's position and size on the screen, in terms of pixels. A **textFormat** element can also be added to control how the timeline text that displays dates will appear.

The *min* and *max* attributes how set the bounds of the time span represented. Setting the play attribute will add a player button. See the section on Timelines in this guide for all of the various options to customize a timeline.

5. How create an XML Data File

The easiest way to import data into the HistoryBrowser is using Excel to create a spreadsheet. The top line should contain the field names and the following lines that data for those fields. For example, this format defines 3 fields, name, sex, age and has 4 people's information:

| name | sex | age |
|-------|--------|-----|
| bob | male | 22 |
| ted | male | 43 |
| carol | female | 33 |
| alice | female | 23 |

Save this out as a tab-delimited text file in Excel by selecting "Save As..." in the File menu and setting the "Save as type" option to "Text- (Tab delimited)" and saving to a file. The GLUE editor has a tool that will allow you to load that file from your computer to the screen area, where it will be formatted into an XML format like this:

```
<TABLE a="name" b="sex" c="age">
  <ROW a="bob" b="male" c="22" />
  <ROW a="ted" b="male" c="43" />
  <ROW a="carol" b="female" c="33" />
  <ROW a="alice" b="female" c="23" />
</TABLE>
```

Click on the "Upload to server" button and that file will be saved on the HistoryBrowser server's data folder using your user id and a name you gave it (i.e. *data/1234-myXMLFile.xml*).

6. How load an XML Data File

An XML data set is loaded into a project's view by way of a **resource** element. That element specifies where the file is (it's *src*), and an *id* to refer to it by when it is loaded from the server.

```
<resource id="myData" type="xml" src="data/1234-myXMLFile.xml"/>
```

Will load a file called from the HistoryBrowser server created in the previous cookbook recipe. (If *src* was set to "http://mysite.org/data.xml", it would load a file called data.xml on the server at mysite.org). That file can have any number of fields and rows.

The data is now accessible to be queried and displayed by its id name. You can access individual elements by specifying them. For example, `status(*myData.name)` would print "bob,ted,carol,alice" on the screen and `status(*myData.name.1)` would print "ted" on the screen, since ted is the 2nd name (the count starts at zero).

7. How to create an animated path

Creating an animated path like animation of Jefferson's letters in the Jefferson's Travels project (<http://www.jeffersonstravels.org>) is relatively easy. You create a project with a base map and a

timeline, then add a **path** to it that contains **dots** that specify that stops. The following example will create a path called *myPath* that responds to changes in time from the timeline by moving an icon between the various dot positions and drawing a red line as it goes.

```
<path id="myPath" wid="10" headStyle="icon:letter" col="0xff0000" tweenLines="true">
  <dot date="1839" x="601" y="168" />
  <dot date="1840" x="1034" y="1083"/>
  <dot date="1841" x="1759" y="959" />
</path>
<glue from="myPath" init="true" />
```

The path element draws a 10 pixels line set by *wid*, and that line is preceded by an icon defined by the *headStyle* attribute. Setting *tweenLines* to true causes a partial line to be drawn between the dots.

Within the **path** element are 3 **dot** elements that actually define the path's course in terms of pixels on the screen. You can find the x and y values by clicking your mouse over the image in the preview and the values will be shown in the bottom-right corner. These numbers will not change if you zoom in and are based on the height and width of the base image.

The *date* attribute determines when that dot will be reached, and responds to the timeline's position as when to be drawn. This can be expressed as year, month/year, or day/month/year.

The final line is a glue element to cause the path to be drawn on the screen. Setting *init* to true will cause the path to be redrawn each time the screen is refreshed, either clicking on the tab or any change in the timeline.

8. Adding Dots via XML to Paths

If your path has a lot of dots, it may be better to put the dots in an xml file and load them once rather than embedding them within the path item. Create an xml file that contains all the information about the dots as if you were embedding them in the path directly. The easiest way to do this is create an Excel spreadsheet with the data.

The first line should contain the field names (lower case only!) and the following lines the dot information, such as this:

| date | x | y | glue |
|--------|-----|-----|----------|
| 1/1765 | 344 | 654 | showInfo |
| 1/1772 | 323 | 444 | showInfo |
| 1/1790 | 244 | 334 | showInfo |
| ... | | | |

Upload and convert the Excel file the server (See the appendix for more information about the process) and fill the path with the dots like this:

```
<resource id="myData" type="xml" src="/1234-MyData.xml"/>
<glue init="true" once="true"/>
  dotfill(myPath,myData)
</glue>
```

The first line creates and loads the xml file we created and uploaded. The glue is set to run only once and uses the **dotfill()** method to fill the path called *myPath*, (which has no dots assigned to it yet) with the data from the *mData* resource.

9. Using routes in animated paths

If you have a number of journeys along a set number of path ways, you can define a collection of dots as a **pathway** and then use that pathway repeatedly at different times. In a normal animated path, you add a **path** element and add the **dots** for each leg of the path. To use a **route**, you set up the **path** element the same way, but instead of adding **dots** directly to the path, we set up an intermediary element called a **pathway**, and add the dots to it. This **pathway** is in turn called upon by a **route** element to be drawn to the screen.

The following script creates a path with two **pathways**, *toEngland*, which contains points on a map of mail to England, and *fromFrance*, which contains points on a map of mail from France. Notice that the dots contain percentages, instead of *times* or *dates* in the other direct method. *The* pct attributes can range from 0-1 and derive the dates from the **route** element later. The first dot is always 0 and the last dot always 1. In the toEngland example because the pct is .5, it will appear halfway and in fromFrance, the second dot will appear 20% into the route.

```
<path id="myPath" wid="10" headStyle="icon:letter" col="0xff0000" tweenLines="true">
  <pathway id=toEngland">
    <dot pct="0" x="601" y="168" />
    <dot pct=".5" x="1034" y="1083"/>
    <dot pct="1" x="1759" y="959" />
  </pathway>
  <pathway id=fromFrance">
    <dot pct="0" x="507" y="156" />
    <dot pct=".2" x="934" y="989"/>
    <dot pct="1" x="757" y="877" />
  </pathway>
  <route pathway="toEngland" start="1/2/1786" end="1/12/1786=" />
  <route pathway="toEngland" start="1/2/1786" end="1/22/1786=" />
  <route pathway="fromFrance" start="3/2/1786" end="3/12/1786=" />
</path>
<glue from="myPath" init="true" />
```

The route element is where it actually is drawn. The *pathway* attribute defines what **pathway** to draw and *start* and *end* attributes define when the whole pathway is drawn. The **glue** element causes the path to be shown on the screen.

10. Adding Routes via XML to Paths

If your path has a lot of routes, it may be better to put the routes in an xml file and load them once rather than embedding them within the path item. Create an xml file that contains all the information about the routes as if you were embedding them in the path directly. The easiest way to do this is create an Excel spreadsheet with the data.

The first line should contain the field names (lower case only!) followed by the route information:

| start | end | pathway |
|-----------|-----------|---------|
| 1/20/1865 | 2/10/1865 | Chicago |
| 1/20/1865 | 2/10/1865 | Detroit |

Upload and convert the Excel file the server (See the appendix for more information about the process) and fill the path with the routes like this:

```
<resource id="myData" type="xml" src="/1234-MyData.xml"/>
<glue init="true" once="true"/>
  routefill(myPath,myData)
</glue>
```

The first line creates and loads the xml file we created and uploaded. The glue is set to run only once and uses the **routefill()** method to fill the path called myPath, (which has no routes assigned to it yet) with the data from the mData resource.

11. How to make booklets for a docviewer

Booklets are a useful way to present information in the HistoryBrowser. Booklets resemble page spreads in traditional books. Booklets contain one or more pages. Each page can have a picture on the left-facing page and text on the right-facing page, a single picture, or text crossing both sides. Each page has the following information:

- **title:** The title of the page
- **src:** The full address of the image (i.e. <http://www.mySite.org/myPic.jpg>)
- **caption:** Caption to appear underneath the picture
- **desc:** The text with any markup you want to add (i.e. bold, italics, links)

In the text you can use the following mark-up tags to control how the text will look:

- **b(text)** will bold the text within the parentheses
- **i(text)** will italicize the text within the parentheses
- **u(text)** will underline the text within the parentheses
- **font(face,size,color)** will size and color the text that follows
- **sp(leading,indent,tabstops)** will set the leading, indent & tabstops for text that follows
- **t()** will add a tab
- **br()** will add a line break
- **link(url,text)** will add a link to url when the text in the parentheses is clicked
- **align(side)** will set the alignment for the text that follows. Can be *left*, *right* or *center*

For example a booklet title of “Martin Luther King” and the caption set to “Martin Luther King Jr. at the Lincoln Memorial”, and the following markup in the desc:

```
font(_sans,11,000000)b(Martin Luther King Jr.) was a i(civil rights leader)
in the 1960s and led many font(_sans,18,ff0000)marches
font(_sans,11,000000)in the southern United States. br()br()He was born in
1929.br()Click link(http://memory.loc.gov,here) for more information"
```

will yield a booklet that looks something like this:



The data source can have 4 fields: *title*, *source*, *desc* and *caption*. Make sure the field names are exactly spelled as those 4 and are in lower case. The *title* field provides a title at the top and a way to select items from the data source. Items with the same title will appear as pages within the document viewer. The *source* field gives a url for a picture if desired, and *desc* is an html formatted text area. If a *caption* field is defined, it will appear underneath the picture.

If both *desc* and *source* are defined, they will appear side by side. If only one is defined, only that one will appear. The text and picture information is supplied by the *filldocviewer* glue method, typically as the result of a query method. The best way to create booklets is by making an Excel spreadsheet where there are columns labeled **title**, **src**, **caption**, and **desc**, like this:

| title | src | caption | desc |
|-------------|---|---------|--------------------------------|
| Chapter One | http://myurl.com | pic1 | The first page in chapter one |
| Chapter One | http://myurl.com | pic2 | The second page in chapter one |
| Chapter One | http://myurl.com | | The third page in chapter one |
| Chapter Two | http://myurl.com | pic3 | The first page in chapter two |
| ... | | | |

A typical script would look like this, where the GLUE editor was used to convert the tab-delimited txt file to XML and uploaded to the server:

```
<resource id="myData" type="xml" src="data/92-tobacc" />
<resource id="viewer" type="docviewer">
  <frame wid="600" hgt="400" left="150" top="50" />
</resource>
<glue id="showInfo" from="viewer" >
  filldocviewer(infoBox,*,myData)
</glue>
```

The 1st line loads that data into a table called “myData.” The 2nd line creates a docviewer called “viewer”. The 3rd line adds a frame to set the sizes and colors of the doc reader. The 5th line adds a glue element called “showInfo” that will display the docviewer on the screen and the 6th line fills the pages of the box with the entire contents of the data in myData.

12. How to create a menu using a *path* container

The following example will show 3 small jpeg files, each containing the chapter name on the screen at the position specified by the *x* and *y* tags. When any one of the jpegs is clicked, the Glue method called *chapterSelect* will be called, and the index of the dot (0-n) is stored in the global variable called *\$\$param*. The *radioshow* method then makes only that one visible by making its alpha 100% and all the unselected ones 0%.

```
<path id="chapters">
  <dot x="50" y="136" style="chap1.jpg" onclick="chapterSelect" />
  <dot x="50" y="164" style="chap2.jpg" />
  <dot x="50" y="192" style="chap2.jpg" />
</path>
<glue id="chapterSelect" from="chapters" init="true">
  radioshow($$param,100,dot)
  [ Do something you want here ]
</glue>
```

13. Making PopUp Information Boxes

Information boxes are popup boxes used to display textual information on demand. They are typically called by clicking on path and graph elements. InfoBoxes can contain a variant of HTML formatting and can be populated using search and replace variable that can be set using a database. The appendix contains detailed information on the text formatting options available.

An **infoBox** is a type of resource of *type* infoBox. It contains the text to be displayed followed by a frame tag that sets the size and look of the infoBox. Since infoBoxes typically popup after a mouse click, the *position* attribute determines the direction from the mouse the box is drawn. In this example, an infoBox called myBox is created that will display some text with two replaceable parameters, \$\$1 and \$\$2. Whenever the **glue** element *showBox* is called, the spot held by \$\$1 will be replaced with *Memphis* and \$\$2 with *New Orleans* for illustration purposes (Normally, these would dynamically come from a dataset of some sort).

```
<resource id="myBox" type="infobox" position="north">
  font(_sans,12,0x990000)May 6, 1831
  font(_sans,11,0x000000)sp(0,0,60)
  Shipped from: $$1
  To: $$2
  <frame backCol="0xFFFFCC" corner="6" wid="200" hgt="150" />
</resource>
<glue id="showBox" from="myBox">
  list($cities,Memphis,New Orleans)
  replacetext(myBox,$cities)
</glue>
```

14. How to Query Data

Being able to query data without needing to send a request to a server is a big advantage in terms of performance. There are 3 parts to querying data: 1). Defining the data set to be queried, 2). Specifying what parts of that data you want to get, and 3). Doing something with the results you get. The results can be ordered by any field and putting in a * in the query portion will result in all fields being returned.

The form of query is **query(listID, dataID, fields, conditions, orderBy)**, where the results of the query are returned in a *listID* from a data set (*dataID*) consisting of the *fields* and rows meeting certain *conditions*, ordered by a field name (*orderBy*).

The *listID* can be an existing list, or the query will create one if it doesn't exist. The *fields* can be an individual field, by name, two or more fields, separated by a + sign (i.e. "name+age"), or a *, which will return all the fields on rows where the conditions are met. The *conditions* determine what rows will be included and contains one or more conditional clause. Each clause consists of a field name, a condition, and a value. (i.e. name EQ John, age LT 20, etc.). Putting a * in the *conditions* place will cause all the data in the data set to be sent to the list.

There are the following conditions possible:

| | |
|-----------|--|
| EQ | Field is exactly equal to value |
| NE | Field is not equal to value |
| LK | Field contains the value with its string |
| NL | Field does not contain the value with its string |
| LT | Field is less than to value |
| GT | Field is greater than the value |
| LE | Field is less than or equal to value |
| GE | Field is greater or equal than the value |

Clauses may be joined by AND, OR and NOT (i.e. name EQ John AND age LT 20 OR sex EQ male). Consider the following example from a prior XML cookbook recipe:

| | | |
|-------------|------------|------------|
| name | sex | age |
| bob | male | 22 |
| ted | male | 43 |
| carol | female | 33 |
| alice | female | 23 |

This query would place "bob" and "ted" in the list called \$myList., ordered by their ages:

```
<resource id="myData" type="xml" src="data/1234-my.xml"/>
query($myList,myData,name,sex EQ male,age)
status($myList)
```

RESULT-> bob,ted

This query would place "bob" and "carol" and "alice" in the list called \$myList:

```
<resource id="myData" type="xml" src="data/1234-my.xml"/>
query($myList,myData,name,age LT 40,age)
status($myList)
```

RESULT -> bob,alice,carol

This query would place "alice" in the list called \$myList., ordered by their ages:

```
<resource id="myData" type="xml" src="data/1234-my.xml"/>
query($myList,myData,name,age LT 40 AND sex NE male,age)
status($myList)
```

RESULT -> alice

This query would place "carol" and "bob" in the list called \$myList., because they both have an o:

```
<resource id="myData" type="xml" src="data/1234-my.xml"/>
query($myList,myData,name,name LK o,age)
status($myList)
```

RESULT -> carol,bob

15. How to make a Control Panel

16. How to Make Graphs

17. How to Make Concept Maps

First, add "cmap" as a new item to "View." Then add an "id" to the Cmap as its title. This "id" will allow the glue you add later to refer back to this cmap. For this example, the "id" is "mccall" to identify the concept map as McCall's relationship map. To block out the background of the base map when you pull up the cmap, add "backCol" from the cmap attributes list and select a cream color. The attributes "cx" and "cy" set the position for the center dot of the cmap. To create a radial map, add a "shape" attribute to the cmap and make the value "radial." The "wid" attribute determines the size of the cmap within the browser's frame.

```
<cmap wid="350" shape="radial" cy="250" cx="400" backCol="0xFFFFCC" id="mccall">
  <dot lab="Archibald McCall" style="icon:person" id="dot0" />
  <dot lab="Tench Coxe" glue="showMe?Tench" id="dot1" />
  <dot lab="War Department" glue="showMe?War" id="dot2" />
  <dot lab="U.S." glue="showMe?Navy" style="icon:person" id="dot3" />
  <line from="" to="dot0" style="line1" />
  <line from="dot0" to="dot1" />
  <line from="dot0" to="dot2" />
  <line from="dot0" to="dot3" />
  <linestyle wid="4" col="0x0099FF" id="line1" />
</cmap>
<glue from="mccall" id="showPhillymap">status(here)</glue>
```

To call up the cmap when a user clicks on a dot on the basemap, first add a "glue" attribute to the dot. For this example, we chose a dot for Philadelphia and labeled this glue "showPhillyMap." That means when you click on the Philadelphia dot, the browser will do whatever the glue "showPhillymap" is programmed for. Next, add a "glue" item to the "View" and, using the "id" attribute, give it the same name you assigned to the dot glue (i.e. showPhillyMap). To connect

this glue to the cmap, add a "from" attribute and enter the cmap "id" as the value. Now when you click on the specified dot, the cmap will pop up.

18. Adding Maps and Vector Graphics

19. Making a Dock

20. Using Files from Flickr

21. Adding Movies

22. Adding YouTube Movies

23. Changing a View via GLUE

Views can also be invisible and not associated with any particular tab. By setting the *visible* attribute to "true" and giving it an *id*, you can use GLUE to cause a view to show within the currently active tab's screen space. If the view is a visible one, the view's tab will be activated.

Assume you had a project that looked something like this:

```
<project>
  <view title="This is View 1" id="myView1" />
  <view id="myView2" visible="false" />
  <view title="This is View 2" id="myView3" />
  <view id="myView4" visible="false" />
  <view title="This is View 3" id="myView5" />
</project>

<glue id="show4">
  setview(myView4)
</glue>
```

There are 5 views, but because the default value of the *visible* is "true" only 3 are visible (myView1, myView3, and myView5). If we called the glue called "show4" from a click or control panel, the contents of the currently active tab's screen would be replaced with whatever we had in the view called "myView4".

24. Floating Concept Map Dots

Dots in a concept map item (cmap) are typically arranged automatically, but you can arbitrarily place a dot anywhere on the screen by setting the *x* and *y* **dot** attributes to a position and setting the **line's** *dir* attribute to float. If you have specified a line style, the line will be drawn from the center of dot specified in the **line's** *from* attribute to the center of the dot.

As an example, here is the tobacco map from the JT2 project. I added a new floating dot (dot13) that hangs off of April (dot4):

```
<cmap id="tobacMap" shape="radial" wid="420" hgt="370" >
  <dot id="dot0" style="leaf.gif" />
  <dot id="dot1" style="jan.gif" lab="January" />
  <dot id="dot2" style="feb.gif" lab="February" />
  <dot id="dot3" style="mar.gif" lab="March" />
  <dot id="dot4" style="apr.gif" lab="April" />
  <dot id="dot5" style="may.gif" lab="May"/>
  <dot id="dot6" style="jun.gif" lab="June" />
  <dot id="dot7" style="jul.gif" lab="July" />
  <dot id="dot8" style="aug.gif" lab="August" />
  <dot id="dot9" style="sep.gif" lab="September" />
  <dot id="dot10" style="oct.gif" lab="October" />
  <dot id="dot11" style="nov.gif" lab="November" />
  <dot id="dot12" style="dec.gif" lab="December" />
  <dot id="dot13" style="icon:letter" x="700" y="300" />
  <lineStyle id="line1" col="0x006600" type="partof" wid="3" alpha="40"/>
  <line from="" to="dot0" />
```

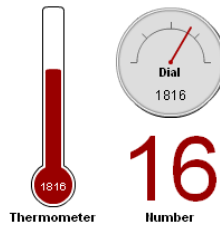
```

<line style="line1" from="dot0" to="dot1" />
<line from="dot0" to="dot2" />
<line from="dot0" to="dot3" />
<line from="dot0" to="dot4" />
<line from="dot0" to="dot5" />
<line from="dot0" to="dot6" />
<line from="dot0" to="dot7" />
<line from="dot0" to="dot8" />
<line from="dot0" to="dot9" />
<line from="dot0" to="dot10" />
<line from="dot0" to="dot11" />
<line from="dot0" to="dot12" />
<line from="dot4" to="dot13" dir="float" />
</cmap>

```

25. Adding Widgets

Widgets are a type of graph that graphically displays a single continuous value on the screen, such as a dial, clock, thermometer, etc. The range of widgets available will grow with time, but they plot the *val* attribute from *min* to *max*. The data is plotted in the color *col*. Here is the script for the following:



```

<resource id="myThermo" type="widget"
sty="thermometer" title="Thermometer"
left="75" top="220" wid="20" hgt="160" min="1804" max="1822" val="1804"/>
<glue from="myThermo" init="true" />
<resource id="myDial" type="widget" sty="dial" title="Dial" left="140" top="220"
wid="100" min="1804" max="1822" val="1804"/>
<glue from="myDial" init="true" />
<resource id="myNumber" type="widget" sty="number" title="Number" left="140"
top="320" hgt="80" wid="100"/>
<glue from="myNumber" init="true">
sub($num,$$curYear,1800)
set(*myNumber.dataVal,$num)
</glue>

```

Appendix

XML Data Format

The easiest way to import data into the HistoryBrowser is using Excel to create a spreadsheet. The top line should contain the field names and the following lines that data for those fields. For example, this format defines 3 fields, name, sex, age and has 4 people's information:

| name | sex | age |
|-------|--------|-----|
| bob | male | 22 |
| ted | male | 43 |
| carol | female | 33 |
| alice | female | 23 |

Save this out as a tab-delimited text file in Excel by selecting "Save As..." in the File menu and setting the "Save as type" option to "Text- (Tab delimited)" and saving to a file. The GLUE editor has a tool that will allow you to load that file from your computer to the screen area, where it will be formatted into an XML format like this:

```
<TABLE a="name" b="sex" c="age">
```

```

<ROW a="bob" b="male" c="22" />
<ROW a="ted" b="male" c="43" />
<ROW a="carol" b="female" c="33" />
<ROW a="alice" b="female" c="23" />
</TABLE>

```

Click on the "Upload to server" button and that file will be saved on the HistoryBrowser server's data folder using your user id and a name you gave it (i.e. `/data/1234-MyXMLFile.xml`).

Web Table Data Import

Existing websites are a great good of data for projects, for example the Historical Census Browser (<http://fisher.lib.virginia.edu/collections/stats/histcensus>) is a great way to get county-level census data from 1790 to 1960. Once you have found a table of data you want, select the entire table and copy (CTRL-C) it into your computer's clipboard.

Paste (CTRL-V) this data in an open Excel spreadsheet. The first line should contain a list of single-word field names that each column can be referred by (i.e. name, sex, age in the precious example).

Save this out as a tab-delimited text file in Excel by selecting "Save As..." in the File menu and setting the "Save as type" option to "Text- (Tab delimited)" and saving to a file.

In the Tools section of the GLUE Editor, select the *Tabbed Data to XML* option. Make whatever change you need to the raw text. Click the *Convert to XML* button. Edit the field names on the first line so that they do not contain any spaces. Click on the "Upload to server" button and that file will be saved on the HistoryBrowser server's data folder using your user id and a name you gave it (i.e. `/data/1234-MyXMLFile.xml`).

Data Import from Many-Eyes

IBM's Visual Communication Lab has a great free website (www.many-eyes.com) for visualizing, storing and sharing data sets. You can automatically pull them in as an xml data source by using a link to its data file. You can import these data sets dynamically into History browser by locating a data set or uploading your own to their site. Click on the link called Data File and use that URL when defining an xml resource. This is a simple example of a data set on ManyEyes:

```

<resource type="xml" id=myData"
  src="http://manyeyes.alphaworks.ibm.com/manyeyes/datasets/things-2/versions/1"
/>

```

Alternatively, you can copy the data on our server by soing the following: Add ".txt" to the url in a web browser, (ie <http://manyeyes.alphaworks.ibm.com/manyeyes/datasets/things-2/versions/1.txt>) and highlight the data manually (CTRL-A) and copy the data (CTRL-C) onto your computer's clipboard.

In the Tools section of the GLUE Editor, select the *Tabbed Data to XML* option. Instead of loading a file to convert, click "Cancel" and paste (CTRL-V) the data over the instructions in the text box. Click the *Convert to XML* button. Edit the field names on the first line so that they do not contain any spaces. Click on the "Upload to server" button and that file will be saved on the

HistoryBrowser server's data folder using your user id and a name you gave it (i.e. */data/1234-MyXMLFile.xml*).

UPLOAD XML PROJECT FILE DIRECTLY

While it is possible to edit the XML directly using the GLUE editor, many people making projects will feel more comfortable editing the XML in a text editor such Oxygen or DreamWeaver. These editors have good undo/redo and context coloring that make the process much easier.

To support this work flow, there is an option in the GLUE editor's File menu called "Upload Local XML File" which will bring up a file box and allow you to select an XML file from your computer's hard drive and upload it to your currently active project. Once uploaded, it will open the same browser window that the "Save and Preview" button uses to preview the project.

The flow goes like this: 1) Edit XML in DreamWeaver 2) Save file to disk in DW 3) Upload Local XML File in GLUE editor 4) See how it looks 5) Go back to step 1.