U.S. DEPARTMENT OF ENERGY

PNNL-22603

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

"Wt" Modular GUI Framework

DES-0047

Revision 1

Charlie Hubbard

June 2012

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

"Wt" Modular GUI Framework

Charlie Hubbard

DES-0047
Revision 1
June 2012

DISCLAIMER

Contents

1 Introduction

The project uses a web-based GUI built using an open-source, C++-based web development toolkit called *Wt*[1]. *Wt* provides a simple to use set of C++ classes for developing arbitrary web-based applications. The toolkit includes an embeddable web server, as well as a rich and ever-expanding set of GUI widget classes. Using the *Wt* toolkit, it is also easy to create custom widgets to suit specific applications.

Using *Wt* components, PNNL staff have developed a modular framework that allows very rapid development of web-based, system control GUIs that interact with the server applications that comprise the project process control software[2].

The foundation of the PNNL GUI application framework is built on three classes:

- `ClientInterface`
- `BaseModule`
- `WebGUI`

The first two of these are defined in the file *wtBase.h* and implemented in *wtBase.cpp*. The third is defined and implemented in *wtWebGUI.cpp*.

The primary documentation for the PNNL GUI application framework is the source code itself and the Doxygen-generated HTML documentation associated with each of the classes and source files listed above. The Doxygen documentation set is automatically built based on the source code itself. It provides the most detailed, most up-to-date descriptions of the code. The document you are reading now is supplemental, and is intended to provide deeper background into the design of the GUI framework. If contradictions between this document and the Doxygen-generated documentation are found, the Doxygen-generated documentation should be considered correct.

2 Theory of Operation

The project's control software is based on our standard client/server architecture, whereby the full software task is divided up into a series of independent, concurrently executing processes, called *servers*, that coordinate their actions by exchanging messages through a standard client message-passing mechanism. One advantage to this architecture is that individual processes tend to be small and very focused toward a particular task (providing the interface to a particular piece of equipment for example).

[1] The Wt project is hosted by the company **Emweb** in Belgium. They also serve as the primary developers. For more information and complete documentation, please see the *Wt* website at `http://www.webtoolkit.eu/wt`.

[2] Project servers all use our standard client/server architecture. Please see design document ***DES-0005, The Client/Server Architecture*** for a complete description of the client/server architecture used throughout the project.

Because servers are so narrowly focused in scope, they are also naturally highly modular. This means, once the initial effort has been expended to develop a new server, odds are likely that that same server can be used again repeatedly on new projects without any code changes at all. This is not just theoretical. On the real-world control systems we've implemented under our client/server model, typically 80% of the servers used on a new project are completely unaltered versions taken from our server code base. This has obvious reliability ramifications, since using pre-existing servers means using tested and proven servers. That they require no modifications also means there's no opportunity to introduce new bugs. This heavy code reuse from project to project also means that the control software for new projects can be put together very quickly. Most of the development effort (writing the individual server applications) has already been done.

In this way, a strongly modular architecture heavily promotes software reuse, reliability, and quick development cycles. These were attributes we wanted for our GUI architecture as well.

2.1 The Skeleton GUI Application (*wtWebGUI.cpp*)

The file *wtWebGUI.cpp* is a skeleton *Wt* GUI application. All real-world GUIs are created by adding to this source file (so for each project, the final GUI is implemented by an application called *wtWebGUI*).

In its initial form, *wtWebGUI.cpp* provides a minimal web page display, and implements certain functionality required by all projects (user authentication, interface locking/unlocking, some system status reporting, and so on). An example of the web page generated by the skeleton version of the *wtWebGUI* application is shown in figure 1.

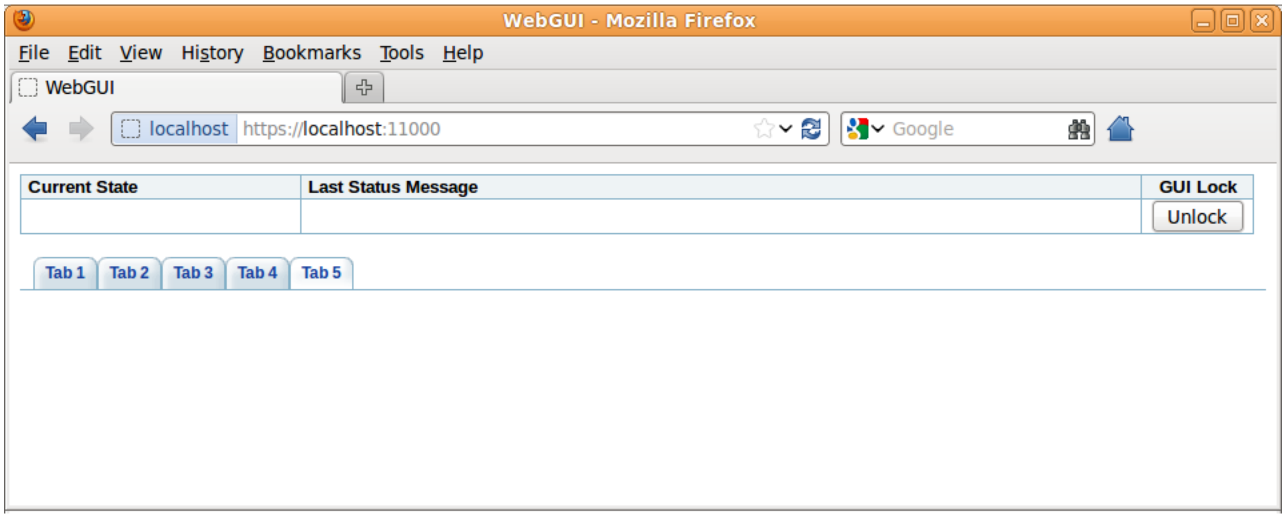

Figure 1: Web page generated by the default skeleton *wtWebGUI* application. For the image, five empty tabs have been inserted for demonstration purposes.

The main feature of the skeleton version is the inclusion of a *Wt* `WTabWidget` on the main page. It is this widget that provides for the framework's modularity. Real-world GUI applications are comprised of a number of separate *modules*. These modules are written in such a way that they can easily be installed into the individual tabs the `WTabWidget` provides. A number of different modules that perform various common tasks (event log viewer, process flow schematic, real-time sensor data plotter, and so on) have already been developed. By-and-large, developing a new GUI consists of grabbing that subset of pre-existing modules that make sense for the current project and installing them into the tabs provided by *wtWebGUI.cpp*.

2.2 The `BaseModule` Class

The specific modules that are installed into the *wtWebGUI's* `WTabWidget` are all derived from a common parent class called `BaseModule`. This base class is defined in *wtBase.h* and implemented by *wtBase.cpp*. The class definition is shown below.

```
class BaseModule : public  Wt::WContainerWidget
{
public:
   BaseModule();
   virtual ~BaseModule();
   virtual void LockModule(const bool lock);
   virtual void TimedUpdates();
};
```

As you can see, there is not much to this class. The primary points of interest are the fact that the `BaseModule` class is derived from a *Wt* widget call `WContainerWidget`, and there are two virtual functions defined that module authors are expected to override with their own implementations.

2.2.1 The *Wt* `WContainerWidget`

Wt's `WContainerWidget` is, as the name implies, a *container* widget. A container widget added to a GUI doesn't display anything on the web page itself. What makes it useful is that any arbitrary set of *Wt* display widgets (buttons, list boxes, labels, check boxes, radio buttons, images, etc., etc.) can be installed into a container widget and organized in any way that makes sense for the module being developed[3]. The individual tabs provided by the *wtWebGUI* `WTabWidget` are themselves container widgets. *wtWebGUI* installs `BaseModule` derived modules (which of course are also container widgets) into its tabs. In the code, this is accomplished using an STL *list* structure called `ModuleList`, defined as follows.

```
typedef list< pair<string, BaseModule*> > ModuleList;
```

[3]Actually, it's even a little more versatile than this. In addition to displayable widgets, container widgets can even include other container widgets. Nesting containers inside of containers is one method *Wt* programmers use for organizing their displays or creating their own custom display widgets.

The `WebGUI` class, instantiates one private member of this type called `moduleList`.

The reader will note that the `ModuleList` structure contains individual elements that are *boost library*[4] style *pairs*. The first item in the pair is an STL string that contains the text label that should be placed on the tab, and the second item contains a pointer to the `BaseModule`-derived module class that's being added to the tab. Once this list is fully populated, the GUI application traverses the list, and creates one new top-level tab for each entry.

The use of a *boost pair* here is unusual and deserves some further explanation. In earlier implementations of the GUI framework, an STL *map* was used to keep track of the modules that were to be installed into top-level tab widget. The map was keyed on a string that contained the tab text, and the payload was simply a pointer to the appropriate module class. This method works fine, but, when traversing an STL map, the entries come back sorted by their key entries. That means the maintainer of the *wtWebGUI* application has no real control over the order in which modules are inserted into the tabs. This is too restrictive. Quite often it is useful to order modules into specific tabs. For example, it might be nice to have the application splash screen module (typically called "About...") installed into the very last tab. Along the same lines, it's useful to put the module that implements the primary user interface into the very first tab (on our projects, this is typically called "Flow Schematic") to make it immediately visible to the user once they've successfully authenticated.

In order to give the application maintainer control over the order in which modules appear in the tabs, an STL list is a better structure. With a list, the tabs are populated in the same order modules were inserted into the list. However, when switching to a list, we lose the map's key field. In the map implementation, the key field held the string that contained the text to be displayed on the tab. That's where the *boost pair* comes in. A *pair* groups two arbitrary items into a single structure that can later be accessed individually using the `first` and `second` operator of the *pair* class.

In the *wtWebGUI.cpp* code, one can find modules being added to the `moduleList` list in the `LayoutSuccessPage()` method. A representative snippet from a real-world version of the GUI code is reproduced below.

```
// Add whatever modules we want for this GUI
moduleList.push_back(make_pair("Flow Schematic",   new FlowModule()));
moduleList.push_back(make_pair("Control Page",     new ControlModule()));
moduleList.push_back(make_pair("Realtime Plotter", new RealtimeModule()));
moduleList.push_back(make_pair("Status",           new StatusModule()));
moduleList.push_back(make_pair("SoH Viewer",       new SOHViewerModule()));
moduleList.push_back(make_pair("Event Viewer",     new EventViewerModule()));
moduleList.push_back(make_pair("About...",         new AboutModule()));
```

This snippet shows how *boost pairs* are created and populated. The code that traverses the list and populates the top-level tab widget with the modules contained therein also exists in

[4]The *Wt* library uses features from the well-known *boost* library extensively throughout its code base, so any system running a *Wt*-based GUI already has *boost* installed. For more information on the *boost* library, please see the *boost* home page at `www.boost.org`.

the `LayoutSuccessPage()` method. The relevant snippet is shown below.

```
// Now add the module contents (one module per tab) to the page
tab = new WTabWidget(root());
for (mlItr = moduleList.begin();  mlItr != moduleList.end();  mlItr++) {
   tab->addTab(mlItr->second, mlItr->first);
   tab->setStyleClass("font");
}
```

This code demonstrates how the individual components of a *boost pair* are accessed and utilized.

2.2.2 The `BaseModule::LockModule()` Method

Because the project GUI is web-based, it automatically supports multiple users connecting to, displaying and working with the GUI at the same time. Some project GUI modules have the ability to directly manipulate system hardware (opening/closing valves, turning on heaters, etc.). Unfortunately, depending on which tab is currently visible, it is far too easy for a user to accidentally make changes to the system's hardware state with a careless mouse click. To avoid this problem, the PNNL GUI framework implements the concept of *interface locking*. This is controlled by the “Unlock” button shown in Figure 1. It works as follows.

When the GUI application first starts up, all modules are *locked*, which simply means their controls are *disabled*. Accidentally clicking on a disabled control does nothing. If a user actually does want to make changes to the system hardware, clicking on the “Unlock” button enables the controls for a small interval (typically five minutes), during which time users can make control changes. Once the interval expires, the GUI automatically disables all controls again[5]. Conscientious users can relock the GUI before the automatic relock timer expires simply by clicking on the button again.

There is a complication with this scheme. In its most naive form, the *wtWebGUI* application could lock the GUI simply by disabling its top-level tab widget. In *Wt* applications, the enable/disable state of a widget trickles down through all of that widget's child widgets, so disabling the top-level tab widget will also automatically disable all of the container and display widgets contained within the tab widget (so, the entire GUI). The trouble is, interface locking is a safety measure designed to prevent accidental changes to the state of the system hardware. Most modules have at least *some* controls that are not associated with hardware state (a button to open a data file browser for example). It doesn't make sense to lock these controls. Only the author of a module knows specifically what controls should be locked and which should be left enabled for his module. This is where the `LockModule()` method comes into play. When a user clicks on the “Unlock” button, the *wtWebGUI* application iterates through `moduleList`, and calls the `LockModule()` method associated with each tab's

[5]Sometimes, especially during initial system development, the automatic relocking feature becomes annoying. If a user *control clicks* (click while holding the *CTRL* key down) on the button instead of just clicking, then the GUI will be unlocked and the automatic relocking feature disabled. Of course, the user can manually relock the GUI at any time simply by clicking the button again.

module, passing to it either *true* (lock the module) or *false* (unlock the module). Every module derived from the `BaseModule` class (which is to say all modules) should reimplement its `LockModule()` method so that it only locks or unlocks those module controls that affect hardware state. Other controls should always remain unlocked.

2.2.3 The `BaseModule::TimedUpdates()` Method

Many or most GUI modules need to update certain display elements—for instance, the reporting values for various sensors at regular intervals. To provide for this, the *Wt* library provides the `WTimer` class. In principle, each module could instantiate its own instance of `WTimer` and use it as it sees fit. However, on a big system, with a lot of modules, this could result in a large number of timer instances running (one timer instance per module multiplied by the number of instances of the GUI that are open in web browsers). Instead, the *wtWebGUI* application instantiates one `WTimer` object set to expire at one-second intervals. With each expiration, the application traverses `moduleList`, and calls each module's `TimedUpdate()` method.

The default version of this method, as provided by the `BaseModule()` class, performs no action. If a module has no need for timed display updates, then there is no reason to overload the base class version of this method. Other modules are free to provide their own versions of the method to update their displays as appropriate. Quite often a module doesn't require updates as frequently as once per second. In this case, it is standard practice to maintain an interval counter, and only act on the call to `TimedUpdates()` once every N number of calls.

2.3 Application Client Interfaces

One additional piece to the GUI framework needs to be considered, and that is the mechanism by which the main *wtWebGUI* application and the modules that are installed into it communicate with the various project servers running on the system.

In the same way that most modules require some sort of timer facility to drive periodic display updates (see the above section), most modules also need to communicate with at least some of the various server applications that comprise the system's control software. As in the timer case, it is certainly possible for the *wtWebGUI* application and each of its display modules to instantiate their own set of client API objects[6]. However, as in the timer case, this could potentially result in a large number of client interfaces being created, creating a lot of duplication.

Instead, the file *wtBase.h* defines a special class that contains one client API object for every server on the system. A global instance of this class, called `clients`, is then created in

[6] For more information on client API classes, see design document ***DES-0005*, *The Client/Server Architecture***.

wtBase.cpp. The *wtWebGUI* application along with all modules installed into it share this one global instance of the client APIs. An example definition for the class is shown below.

```
class ClientInterface {
public:
   ClientInterface();

   AnalogClient    analog;
   ControlClient   control;
   DigitalClient   digital;
   LoggerClient    logger;
   PIDClient       pid;
};
extern ClientInterface clients;
```

Real-world GUIs will need to add or remove client interfaces from this class as necessary to match their specific project.

3 Starting the GUI

When the GUI application (*wtWebGUI*) is compiled, it is linked against the Wt-provided *libwthttp* library. This provides it with an embedded web server capable of communicating with multiple clients (web browsers) at the same time using either standard *http* or *https* connections (the latter being encrypted). The application is started from the command line (or, more typically, from a shell script). Various command line parameters exist for setting options on the web server. In this section, we'll look at typical command line options. A complete list of options supported by the embedded server can be found on the Wt website at http://www.webtoolkit.eu/wt/doc/reference/html/InstallationUnix.html.

3.1 HTTP Access

The GUI application is most easily configured to be accessed via regular HTTP. The following command line will do the trick:

```
./wtWebGUI --http-address 0.0.0.0  --http-port 80  --docroot .
```

The specified command line options tell the embedded web server to allow connections on TCP port 80 of any of the computer's network interfaces (address 0.0.0.0). The *docroot* parameter specifies the directory the embedded server should consider its “root” directory when accessing static files.

3.2 HTTPS Access

The GUI application can also be configured to accept and communicate over HTTPS connections. This is advantageous for security reasons, because HTTPS connections are encrypted from end to end.

Setting up for HTTPS access is somewhat more difficult. It has a more complex command line, but the real complication is the need to generate a private server key file signed by a certificate authority and a file containing random Diffie-Hellman parameters used for performing the link encryption.

For internal use, you can create your own signed private server key file using the *openssl* package for your particular Linux distribution. To create the Diffie-Hellman parameters file...

```
openssl dhparam -check -text -5 512 -out dh512.pem
```

This generates a file called *dh512.pem*. This file should exist in the directory from which the GUI application is started. Creating the signed server key file is somewhat more complicated...

```
openssl genrsa -des3 -out server.key 1024
openssl req -new -key server.key -out server.csr
cp server.key server.key.org
openssl rsa -in server.key.org -out server.key # removes the passphrase
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
cat server.crt server.key server.crt > server.pem
```

This results in a file called *server.pem*. Like the *dh512.pem* file, this file should exist in the directory from which the GUI application is started. Also, the access permissions on the *server.pem* file need to be such that the owner of the GUI application has read access to it, and no other users can access it at all. This is typically done as follows.

```
chmod 400 server.pem
```

These two files (*dh512.pem* and *server.pem*) only need to be generated one time. After that, the GUI application will use these same two files every time it is started. To configure the application's embedded web server to communicate using HTTPS *only*, the following command line is appropriate (shown here split into multiple lines simply to fit well on the page).

```
./wtWebGUI --https-port 11000 --https-address 0.0.0.0  --ssl-certificate=server.pem \
          --ssl-private-key=server.pem --ssl-tmp-dh=dh512.pem --docroot . &
```

If you want the embedded web server to support both types of connections, you can add in the "http" command line parameters from the previous section as well.