Journal of Physics: Conference Series **513** (2014) 012009 doi:10.1088/1742-6596/513/1/012009

FwWebViewPlus: integration of web technologies into WinCC OA based Human-Machine Interfaces at CERN

Piotr Golonka[1], Wojciech Fabian, Manuel Gonzalez-Berges, Piotr Jasiun, Fernando Varela-Rodriguez

CERN: European Organization for Nuclear Research, Geneva, Switzerland

E-mail: Piotr.Golonka@CERN.CH

Abstract. The rapid growth in popularity of web applications gives rise to a plethora of reusable graphical components, such as Google Chart Tools and JQuery Sparklines, implemented in JavaScript and run inside a web browser. In the paper we describe the tool that allows for seamless integration of web-based widgets into WinCC Open Architecture, the SCADA system used commonly at CERN to build complex Human-Machine Interfaces. Reuse of widely available widget libraries and pushing the development efforts to a higher abstraction layer based on a scripting language allow for significant reduction in maintenance of the code in multi-platform environments compared to those currently used in C++ visualization plugins. Adequately designed interfaces allow for rapid integration of new web widgets into WinCC OA. At the same time, the mechanisms familiar to HMI developers are preserved, making the use of new widgets "native". Perspectives for further integration between the realms of WinCC OA and Web development are also discussed.

1. Context and problem specification

The screens of operator consoles for many of the control systems at CERN display highly specialized applications built on top of the *Simatic WinCC Open Architecture* commercial SCADA product (formerly known as *PVSS*), from ETM/Siemens. These *Human-Machine Interfaces* allow users to supervise and command complex systems through applications containing push-buttons, check-boxes, list-views, but also non-standard objects such as trend-plots or specialized bar-graphs (see Figure 1).

Engineering of these applications (*synoptic panels*) is performed using rapid development environment, called *GEDI*: the elements of the user interface (*widgets*) are instantiated and configured with a drag-and-drop interface and property editors. The elements are connected to data sources and animated with business logic, using a dedicated scripting language called *CTRL* .

The widgets available for engineering, may be extended through a custom C++ plugin mechanism called *EWO* (External Widget Object), which provides highly customized and optimized interactive graphical objects, and integrates them into GEDI for engineering. However, being a binary interface, EWO requires development and maintenance effort: every upgrade to a new versions of WinCC OA requires the plugins to be recompiled and the code often needs adjustments.

[1] To whom any correspondence should be addressed.

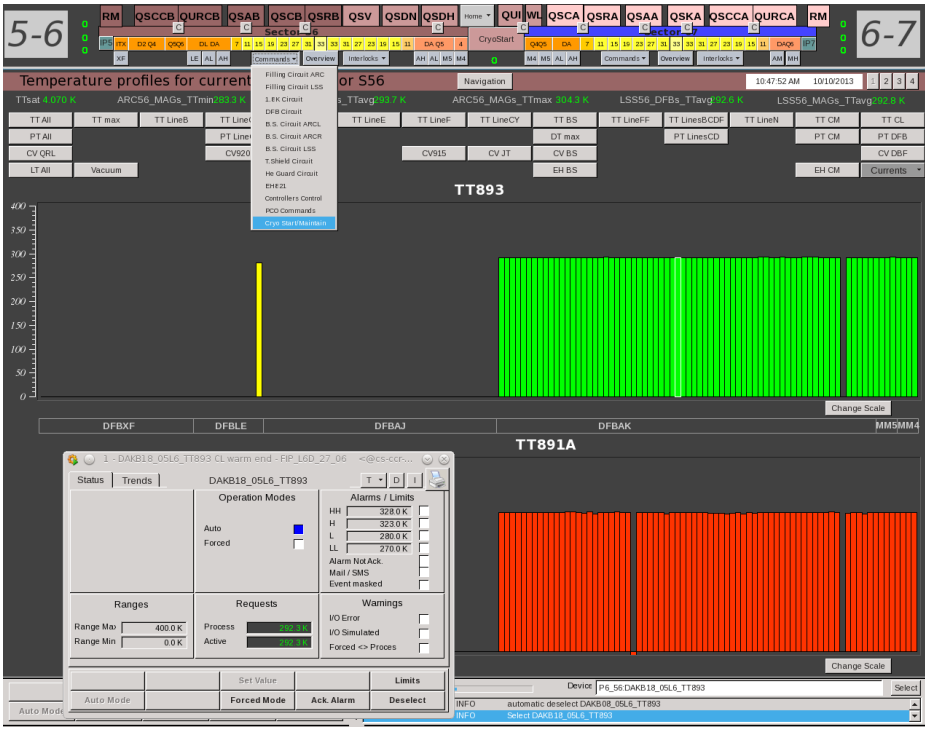

Figure 1. Example of a Human-Machine Interface display for the control of LHC cryogenic system at CERN, showing the custom bar graph visualization plugin.

To simplify the maintenance and rapidly deliver new widgets with less effort, we set up a project to identify a technology that could be used for future visualization extensions of WinCC OA. Technically, we searched for a solution providing a common platform-neutral graphical engine, programmable with a high-level scripting language with performance and quality adequate to visualize live data. Our ultimate aim would then be to re-implement the functionality of numerous existing binary plugins and to enhance the user experience by deploying high-level scripts, executed by the plugin, which will be implemented using the new technology. The level of code abstraction would therefore be elevated and development of visualizations made easier, and only a single common binary EWO would need maintaining.

2. Technology selection

In addition to the main requirements, the proposed solution had to meet additional goals. It had to be mature and well supported to be integrated into production systems, with a clear future evolution path in line with observed software trends. The supported platforms need to include Windows and Linux, and integration into mobile devices would be an asset.

After having evaluated technologies such as fw3DViewer [1], ROOT [2] and QtQuick/Qt3D component of Qt[3], we turned our attention to the QtWebKit[4] generic web-rendering engine readily available in Qt. The initial evaluation performed with the simple *WebView* EWO, distributed with WinCC OA convinced us of its suitability for our needs.

QtWebKit is based on the widely known WebKit[5] open source project, which powers web browser such as Chrome or Safari. It employs widely adopted web technologies such as HTML, HTTP, QML, CSS and SVG. Its graphics engine provides the standardized, platform-neutral Canvas element of HTML5, with a clear path for extension towards 3D graphics (through the soon-to-come WebGL). Programming capabilities are provided by the highly-optimized interpreter of JavaScript,

which is a widely known scripting language throughout many domains, not just within the web browser universe. The use of widely used web engine reduces cross-browser incompatibilities as well.

Ultimately, a web-based renderer enables the reuse of impressive number of web visualization widget libraries (JQueryUI, MooWheel, Google charts, etc), available with permissive licensing. In addition, the exercise of implementing a custom widget from scratch demonstrated a significant reduction in code complexity and development effort (see Chapter 4).

3. Features and integration

FwWebView plus was implemented as a standalone self-containted web-rendering plugin ready to be embedded directly into WinCC OA panels, concurrently with other widgets and EWO plugins.

The use of the web-rendering widget implicates the need for hybrid programming: in addition to WinCC OA's *CTRL* scripting language, data types, and widgets, the technologies required by the web, namely *JavaScript* and *HTML* need to be employed to maintain the actual content displayed by the widget, and also handle user interaction. Successful integration of a new web widget through fwWebViewPlus, typically requires some code to be developed on both sides. In what follows we will refer to JavaScript/HTML code, that is executed by the web rendering engine, as development in the *web context*, and the development on the native WinCC OA side as *CTRL context*.

The aspect that differentiates fwWebViewPlus from the WinCC OA's own WebView EWO is its deep integration with native mechanisms and conventions, making the necessary hybrid programming more intuitive and clear. In this chapter, we will describe the features making the application-engineering with web widgets as rapid as if one uses native user interface elements.

To unify the interface between the two contexts we followed a well known URI concept, and defined new *schemes*[2]. The *project:* scheme implements the look-up of files in the hierarchy of folders and provides consistency with standard file specification and over-parameterization mechanisms. The *wincc:* scheme delivers a common low-level method to communicate with *CTRL* context from JavaScript code, which in turn is used to implement the remaining integration features.

To assure best performance we made it possible to access the control process data (so called *datapoints*) directly from JavaScript code, with no intermediate processing by the CTRL engine. The code in the web context may retrieve the current values on demand (*dpGet*) or register JavaScript callback functions hooked to value-change events (*dpConnect*).

Readability of the web-context code is improved by the use of a library of high-level JavaScript functions (called *the API*), which hides the complexity of *wincc:* URIs. Care was taken to make the syntax of the functions familiar to the programmers accustomed with CTRL. The API, as well as copies of other handy JavaScript libraries such as *JQuery*, were embedded into fwWebViewPlus and made accessible through the *resource:* URI scheme, hence making it unnecessary to maintain their copies separately. Incorporating updates for resources is straightforward and may be applied to production applications in a centralized way by simple redeployment of fwWebViewPlus component.

Access to the web context from CTRL is provided by methods of the EWO object operating directly on the HTML document content. Asynchronous execution of any valid JavaScript code may also be requested: existing JavaScript functions could be called, and the input and output parameters are automatically converted to their native representations: mapping type in CTRL, and JSON-serialized JavaScript objects in the web context. It is also possible to register a JavaScript function to be recognized by the CTRL context as if it was a native method of the EWO. This allows the exposure of elegant interfaces for interfaced web widgets; dynamically registered methods are recognized by the run-time and also by the engineering part (CTRL script editor).

Implementing interactivity for web widgets requires the ability to emit asynchronous notifications from JavaScript, which in turn trigger a callback function in the CTRL context. The *emitSignal()*

[2]Scheme is the top level part of the URI, followed by the colon character, for instance *http:* or *ftp:* or *file:* .

method of the API provides the necessary mechanism; for instance, one may pass mouse-click, keyboard events, or web socket notification received by JavaScript to the CTRL context.

The most used settings of the EWO are exposed to the property editor of GEDI making them easy to change; similarly WebKit notifications, such as *onPageLoadFinished* are available in the list of available CTRL event callbacks. The property values are in turn made accessible, via the API, to the web context; for instance the background colour property may be set in the property editor, using GEDI's colour-picker, then decoded by JavaScript and used to alter the rendering of HTML document.

The security is also taken care of: dedicated EWO properties allow the developer to disable selected URI schemes: deactivating *http(s)* scheme(s) disables access to any non-local resources or code; similarly deactivating the *wincc* scheme disables direct access to datapoints from JavaScript.

Finally, the error and debugging streams of JavaScript are redirected to standard WinCC OA's logging streams to assure consistent diagnostic and debugging facilities.

The ultimate engineering experience that we aim at is a complete integration of pre-packaged web-widgets (e.g. charts or graphs), such that their use in new synoptic panels and connecting to data would be achievable through trivial drag-and-drop and property-editing operations.

We observed that the performance of web widgets is slightly worse than for native C++ codes, yet significantly faster than those implemented using graphical primitives and CTRL. For the visualizations we currently maintain, this performance should be sufficient.

4. Examples of use

Visualizing the connectivity state of nodes in large distributed control systems, based on real-time data was the first application of fwWebViewPlus. The fwNetVis application, currently being deployed at CERN, integrates MooWheel[6] JavaScript widget to present information through an interactive graph, see Figure 2, making it easy to identify root-causes and correlate symptoms of connectivity faults. By reusing the existing web widget it was possible to rapidly deliver a new visualization.

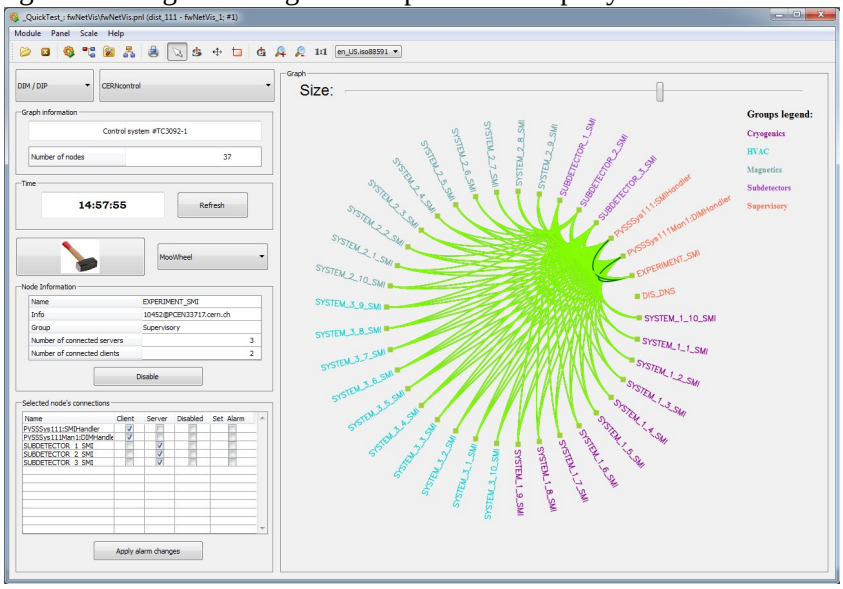

Figure 2. fwNetVis application showing a interactive graph of connections

Another kind of visualization not available in WinCC OA's widget set, yet frequently demanded by application engineers, is a *heat map*: a frequently used graphical representation of data contained in a matrix, where colours are used to represent the values. To exercise the concept of developing a brand new visualization with web technology and assess the effort for initial code development effort and maintenance we implemented a generic heat-map visualization in HTML5/JavaScript and integrated it

into a test application using fwWebViewPlus, see Figure 3. The widget was animated with live data and we observed satisfactory performance (display of data within a 100x100 cell grid with the rate of change of a few frames per second) and allowed for interactive inspection of values (in a tool-tip window). The complete JavaScript implementation fitted into 450 lines of code, which could be used not only with WinCC OA, but also in web applications.

To explore the flexibility of fwWebViewPlus we also implemented a proof-of-concept of a complete functional panel implemented solely with HTML/JavaScript. A simple value monitor dynamically connecting to a specified datapoint, and showing the changes was implemented within 50 lines of HTML, employing JavaScript for animation and connection to data and CSS for visually attractive styling and layout.

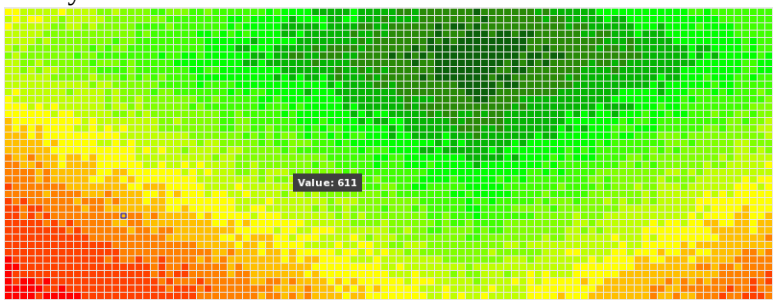

Figure 4. Heat map visualization widget implemented from scratch, in JavaScript.

5. Outlook and conclusion

We presented a plugin for WinCC OA user interfaces allowing the embedding of JavaScript/HTML visualizations in the applications used in CERN control rooms. We achieved seamless integration with WinCC OA's engineering environment to make the web-CTRL hybrid programming straightforward and intuitive. We also demonstrated the use of fwWebViewPlus in a real diagnostic application used at CERN and showed that integration of new visualizations based on web technologies allows for easier development and maintenance. It will soon be released as a standard component of JCOP Framework.

The approach we presented is complementary to the currently observed trend in making existing applications available on the web. There is however a potential for convergence: fwWebViewPlus may allow WinCC OA engineers to familiarise with web technologies and apply them in their WinCC OA synoptic panels. We strongly believe that future migration of such web-enabled panels towards fully web-based applications will be significantly simplified and straightforward.

In addition to providing new visualizations, we also plan to explore new possibilities provided by JavaScript data processing and visualization packages such as *D3.js* or *Processing.js* and look at the new opportunities of unleashing the power of JavaScript through projects such as *Emscripten* C++ to JavaScript compiler. With WebGL becoming standardized and supported in Qt-5, interactive 3D graphics will also become available for fwWebViewPlus in the near future.

References

[1] Golonka P, Gonzalez-Berges M (2009) Towards 3D Human-Machine-Interfaces: Generic "3DViewer" Extension for the Control Systems Displays at CERN *Proc. ICALEPCS 2009 (Kobe)*, https://cern.ch/enice/JCOP+Framework+3DViewer
[2] ROOT Object-Oriented framework for data analysis, http://root.cern.ch
[3] Qt cross-platform application framework http://qt-project.org
[4] QtWebKit component (formerlyWebView) , http://qt-project.org/doc/qt-4.8/qtwebkit.html
[5] WebKit open source web browser engine, http://www.webkit.org/
[6] MooWheel: a JavaScript connections visualization library, http://labs.unwieldy.net/moowheel/