Upgrading the Interface and Developer Tools of the Trigger Supervisor Software Framework of the CMS experiment at CERN

Glenn Dirkx, *Student, KU Leuven, glenn.dirkx@student.kuleuven.be*
Dr. Christos Lazaridis, *University of Wisconsin, Christos.Lazaridis@cern.ch*
Dr. Peter Karsmakers, *KU Leuven, peter.karsmakers@kuleuven.be*

***Abstract*—The Compact Muon Solenoid (CMS) Trigger Supervisor (TS) is a software framework that has been designed to handle the CMS Level-1 trigger setup, configuration and monitoring during data taking as well as all communications with the main run control of CMS.**

The interface consists of a web-based GUI rendered by a back-end C++ framework (AjaXell) and a front-end JavaScript framework (Dojo). These provide developers with the tools they need to to write their own custom control panels.

However, currently there is much frustration with this framework given the age of the Dojo library and the various hacks needed to implement modern use cases.

The task at hand is to renew this library and its developer tools, updating it to use the newest standards and technologies, while maintaining full compatibility with legacy code.

This paper describes the requirements, development process, and changes to this framework that were included in the upgrade from v2.x to v3.x.

***Index Terms*—CERN, CMS, L1 Trigger, C++, Polymer, Web Components.**

I. Introduction

The CMS experiment at the European Organization for Nuclear Research (CERN) consists of many components. One of them is the Level-1 (L1) trigger, designed to filter the enormous amount of data generated by the proton-proton collisions at the experiment (currently around 80TB/s[1]). The Trigger Supervisor (TS) is a software project that aims to control the L1 trigger. This includes setup, configuration, and monitoring before and during during data taking. It allows for controlling various aspects of the L1 trigger using panels in a web interface. This interface is custom built for each use case, although some generic panels exist (commands, operations, ...).

The main software library that facilitates this, is AjaXell. It provides the developer tools to make a custom panel. It does, however, have a few problems.

Firstly, the Dojo library that AjaXell uses to render the panels is old. It misses functionality required for modern use cases and it even starts to break down on modern browsers. For example, a modal dialog does not render in Google Chrome 44 but the transparent background that usually forces the user to make a choice in the modal does render. This effectively blocks the user from doing anything and forces the user to reload the page and start over. Today, when a developer wants to provide new functionality, the solution is to just manually write the HTML and JavaScript.

Secondly, the current state of affairs requires developers to write everything in one big C++ file. Panels consist of many languages (C++, HTML, JavaScript, and CSS) and combined with the fact that much functionality is written manually, this results in very messy and unreadable code. Several existing panels are very difficult to modify because of this.

This paper describes the changes that have been made to the TS to solve these problems.

P. Karsmakers is with the Department of Electrical Engineering (ESAT) TC, KULeuven, Technology Campus Geel, Belgium, e-mail: peter.karsmakers@kuleuven.be

II. Related Work

The full original design of the Trigger Supervisor framework can be consulted in the PHD thesis of Ildefons Magrans de Abril[2]. This thesis describes both the hardware and software design decisions that were made and provide more detail about the TS in general, as this paper merely describes the operator interface redesign.

It's also recommended to read the Phase II upgrade technical proposal of the CMS experiment[3] as it describes the upgrade from Phase I [4] and its new architecture and ideas that the work described here accompanies.

III. Functional Requirements

Ideally the result would be a new framework, much more powerful than its predecessor, that yet manages to achieve 100% compatibility with legacy code.

The main objectives are cleaner code, better maintainability, better documentation, and easier development. Making the framework easier to develop on, will invite developers to write more advanced code.

IV. Upgrade options

Although in the final stage there will be no more legacy code, old code must still remain functional in the new environment to allow for a smooth transition.

This limits the available options at the back-end. Because of this, only extra code to the existing C++ codebase can be added. Changes are not possible.

On the front-end side there are a bit more possibilities. The only important requirement is that whatever the new code looks like, the old Dojo code must be able to run alongside it.

1) Dojo v1.10: It is not possible to just upgrade to a new version of Dojo. Currently AjaXell uses Dojo 0.4 and starting from Dojo 0.9 there has been a major API change. Two different versions of Dojo cannot run concurrently since they still share a lot of function calls.

Also this approach would not solve any of the currently existing problems. Interfaces would still have messy code and frustrated developers.

2) Web Components: Web Components[5][6] are additions to the HTML5 standard. They enable a developer to develop custom HTML tags, the idea is to mitigate the 'div soup' problem[7] where the web application's source code increases exponentially in size as the complexity of the app increases.

This standardizes an approach seen in many modern JavaScript frameworks such as AngularJS, Ember.js, Knockout.js, Dojo, and Backbone.js. These all allow a developer to declare specialized 'elements' in order to make developing a smart web application easier. However, by relying on the Web Components standard it can be safely assumed the problem encountered with Dojo 0.9, which introduced breaking API changes, will not occur again. Despite being a new standard, support for all CERN-supported browsers (firefox ESR 24-current) can be achieved using the webcomponents.js polyfill.

3) Polymer: Polymer is a relatively new library, built directly on the Web Components standards, developed by Google. It represents the way Google thinks Web Components should be used. The reason Polymer is very useful is that it has the potential to allow us to introduce proper Separation of Concerns (SoC) principles (V-A) to the development environment.

V. The new development environment

A. Separation of Concerns

SoC is a design primitive, dictating a modular design of software. This has been implemented in three ways.

Firstly, different syntaxes now are housed in their own files. This allows for significantly less messy code and enables us to implement specific optimizations for each language (for example a CSS pre- and post-processor).

Secondly, the developer is not limited to one source file for each syntax. If circumstances would make some code easier to manage if it is housed across multiple files this is now possible. An example of this would be a panel with multiple specialized sections. Separating these sections will make the code easier to read and maintain.

Thirdly, this approach pushes developers to separate data from markup. This is a very good thing as it causes the code to once again be much more readable. By having the C++ code only produce the necessary data and putting all rendering and interaction on the front-end a developer can also safely replace rendering logic or user interaction flow without having to worry about data generation.

B. Build Cycle

Instead of loading all the separated files individually at runtime, they will instead be compiled together at compile-time. This will improve loading speeds. The tool used to do this is Grunt (http://gruntjs.com/), a task runner built on nodeJS that is used to compile, minify, lint, unit-test,... front-end code languages. It has very wide community adoption, which results in a very rich set of tools available for use.

C. Code optimization

Now that every code language is housed in specialized files some optimizations can be done on them at compile-time. The main objective of these optimizations is to achieve as much browser compatibility as possible.

1) JavaScript: In order to ensure compatibility with all required browsers all JavaScript code is transpiled by Babel (https://babeljs.io/). This will ensure that newer syntax, like ECMAScript 2016 (ES7), will be transpiled into a more compatible equivalent.

Also the JavaScript code will be transpiled by UglifyJS (https://github.com/mishoo/UglifyJS). This will implement various code optimizations[8] making the code faster.

2) CSS: Developers are given the possibility to write SASS code, an extension of the CSS syntax, that will be transpiled into CSS on compile-time using libsass (http://sass-lang.com/libsass).

Also Grunt will automatically add vendor-specific prefixes to CSS properties to maintain the required browser compatibility using a tool called autoprefixer (https://github.com/postcss/autoprefixer).

D. Code sharing

Code duplication should be minimized as much as possible. Code that is used frequently is therefore moved to a separate code repository available for anyone to use. These include things like chart libraries, layout frameworks, and some in-house components such as an auto-updater.

VI. Documentation

Documentation is something commonly taken too lightly. Fortunately there are some tools not only to make good documentation but also to encourage developers later on to write proper documentation.

Most of the documentation is housed along with the source code itself. The goal is to minimize separation of code and documentation as this easily leads to inconsistencies between code and documentation.

A. Inline Documentation

Advantages of inline documentation are the reduced chances for outdated documentation and being able to enrich source code with typed annotations [9].

Source code consists of C++, JavaScript, HTML, and CSS code. The inline documentation described here is applicable to the last three.

The syntax used to document JavaScript code is called JSDocs and is currently at version 3[9][10]. It provides us with a rich set of expressions enabling a developer to write documentation comparable to JavaDoc.

In addition there are specific points in the source code where a developer can provide code examples and extra directives to document HTML and CSS code. This is however a non-standard method since there is no standardized way to inline-document any of the other languages.

B. Global level

The global level is the only level where documentation is separated from the source code. This houses documentation aimed to teach users and developers the concepts and ways of thinking regarding this codebase. It teaches developers the basics of the structure they will be developing in and the philosophy behind this structure.

This global documentation level is built using Sphinx (http://www.sphinx-doc.org/) and provides a single point of entry for people looking for documentation and will guide readers to the next level of documentation when they are ready for it.

C. Package level

The codebase is composed of a number of packages. Each package automatically generates documentation describing its content and capabilities.

This documentation is generated in the Grunt build cycle described earlier. It loads and interprets every component of the package and generates a summary page giving a general overview and pointing to several useful resources for each component such as the documentation on the element level, a link to the code repository, and a link to a live demo of the component if available.

The code it uses to render this documentation is housed in the source code of each component. It gets interpreted by Grunt and is then compiled in the package documentation page.

D. Element level

The lowest level of documentation is documentation of individual web components. This level is also auto-documented from the component's source code. But unlike the documentation on the package level, where documentation is generated on compile time, the documentation here is rendered on the fly.

This is done by using a specialized web component called 'iron-component-page' (https://elements.polymer-project.org/elements/iron-component-page). It interprets the source code of the component and comments left by the developer and compiles this into a documentation page.

This documentation provides an overview of all the properties and available calls of this component. It can also provide code examples and even live demos.

	TS 2.1.0	TS 3.4.0	difference	difference (%)
Loading	44.5ms	112.4ms	+67.9ms	+152.58%
Scripting	1227.6ms	1187.6ms	-40ms	-3,26%
Rendering	29.7ms	171.0ms	+141.3ms	+475,76%
Painting	7.5ms	36.1ms	+28.6ms	+381.33%
Other	106.4ms	335.9ms	+229.5ms	+215,69%
Idle	213.6ms	775.1ms	+561.5ms	+262,87%
Total	**1.63s**	**2.62s**	**+990ms**	**+60,73%**

TABLE I
PAGE LOADING TIMES FOR TS 2.X AND 3.X

VII. RESULTS

A. Loading times

Table I shows an overview of the initial full page loading times for the legacy TS (version 2.1.0) and the new TS (version 3.4.0). That is, a page load from a new browser tab with all caches removed.

This test is performed with the timeline panel of Google Chrome 50.0.2661.86 (64-bit).

It is expected that the TS 3.x has higher values for everything in this table, because it loads two front-end libraries (Dojo & Polymer).

Notable is the decrease of scripting time for the TS 3.x relative to the TS 2.x. This is because Dojo is minified and packaged into one JavaScript file in the TS 3.x release, where as in the TS 2.x release it was not. Also, because this test is performed in Google Chrome, which has native support for Web Components, very little scripting needs to be done. This result will be different in other browsers like Mozilla Firefox, where Web Components support needs to be emulated. Then again, the lazy loading system largely removes this overhead from the initial page loading time, so only minor differences would be expected here.

Rendering time has increased the most going from TS 2.x to TS 3.x. This makes sense as Polymer renders everything on the front-end, whereas Dojo used to render everything server-side. During initial page load this rendering load is primarily caused by the rendering of the left side menu. The increase of painting time follows the same logic as the rendering time.

Also notable is the increase of idle time. This means that the browser needs to wait for a task to finish before it can start another. This is caused because the TS 3.x loads the default panel after the initial page load. Which means the TS makes extra network request, to fetch an interface panel, right after initializing. This is counted with the initial page load. TS 2.x just shows a blank page, it loads no default panel. Because the browser needs to wait for the extra network requests to finish before it can render the default panel, the idle time goes up by a lot.

In total, the initial page loading time increased with about 60%, which is an acceptable increase given the new TS runs 2 libraries concurrently.

B. CPU consumption

Both TS releases have negligible CPU usage when doing a fresh page load, and stay at 0% CPU usage when the user is not interacting with the system.

	Google Chrome	Firefox
TS 2.1.0 (Dojo)	20.051MB	7.06MB
TS 3.4.0 (Dojo + Polymer)	24.564MB	10.96MB
difference	+4.513MB	+3.9MB
difference (%)	**+22.51%**	**+55.24%**

TABLE II
MEMORY USAGE FOR TS 2.X AND 3.X IN MOZILLA FIREFOX AND GOOGLE CHROME

TS 3.x uses hardware acceleration for it's animations since they are all made using CSS transform properties or using Web Animations[11]. The only exception to this is the 'paper-spinner' element. Which displays a loading animation. The TS 2.x release did not have any animations.

C. Memory consumption

The Dojo library of TS 2.x contained memory leaks, and could lead to a web browser using an excessive amount of memory when an interface was used for a long duration of time.

Unfortunately, legacy panels in the new TS still suffer from this memory leak. This is because the circular references causing the memory leak reside in the Dojo library itself, and thus would be impractical to address. Therefore, any interface that included auto-refreshes had the highest priority to be converted to a new TS 3.x interface.

Because TS 3.x uses client-side interface rendering rather than server-side as the TS 2.x did, it uses more memory from the browser.

In TS 3.x the memory used by an interface panel will be released after there is a switch to another panel. It is also known that in TS 2.x the memory consumption grows linearly with the amount of panels loaded by the user.

To test the difference in memory consumption, both TS versions were opened in a new tab while memory consumption is monitored. No panels are loaded, the interfaces are just left for 120s. The mean memory consumption in those 120s is then taken as the mean memory consumption for that TS release. The results of this test are shown in table II.

VIII. FUNCTIONALITY

TS 3.x has functionally more capabilities for the interface than TS 2.x had. More importantly, the TS interface is now no longer bound to one framework. Any Web Component can be used, and extra functionality can be developed in-house. This unlike TS 2.x where developers were functionally bound to the elements the Dojo developers provided.

This makes TS 3.x far more easy to change, and thus more ready for the future.

IX. SDK IMPROVEMENTS

The fact that multiple programming languages are no longer placed into one file, but distributed across multiple files, makes the developing an interface panel a lot easier.

The Web Components approach to build interfaces gives developers a set of powerful tools that are easy to use and extend.

X. DEVELOPED PANELS

The Control Panels are a set of custom interfaces, developed for an individual cell. The other panels however occur on every cell. And are upgraded as part of the new TS release.

A. Commands

The new commands panel use the 'command-input' element for its input. Making it easily extendible to understand more input types (e.g. vectors). Currently it understands number, int, long, unsigned int, unsigned long, short, unsigned short, string, double, and float input.

B. Operations

The TS 2.x operations panel had some problems with auto-updating. The state diagram tended to update very late, if it updated at all. Result data and new available commands usually took more than 10 seconds to show up in the interface.

The new operations panel is now far more responsive. The state diagram is available when clicking on an icon, as it was deemed a waste of space to show it by default.

C. Flashlists

The flashlist panels now have a user-configurable auto-update function. The flashlist can deploy custom renderers in the table depending on the data type, for example a date will be shown as relative time (e.g. 9 minutes ago), instead of just showing a time stamp. This list of custom renderers can be extended easily.

XI. CONCLUSION

The main objective was to upgrade the TS to be able to provide more advanced interfaces, and to keep compatibility with legacy interfaces.

The new interface engine has achieved 100% backwards compatibility, while providing a completely new way to develop new interfaces.

This new interface engine and can be easily extended and is ready for any future use-cases as it is built to change. The developers are not bound to the functionality of one framework, rather it is build on open standards and thus ensures maximum compatibility with future technologies.

The interface developers now have internal, semi auto-generated, documentation at their disposal and have an active community on the world wide web to fall back to.

ACKNOWLEDGMENT

I would like to thank the following people for their assistance during this project:

Christos Lazaridis for being a great mentor and for not getting mad when I break the nightlies or even SVN itself.

Alessandro Thea for his advice on how to proceed with implementing new functionalities and

his supply of motivation and inspiration.

Evangelos Paradas for his guidance trough the architecture of the TS and pointing me to useful resources.

Simone Bologna for his enthusiasm and patience when finding bugs, and his steady supply of ideas.

Furthermore I would like to express my thanks to the entire Online Software team for the freedom and trust I've been given that allowed this project to get as far as it has.

REFERENCES

[1] T. C. Collaboration, "The cms experiment at the cern lhc," *Journal of Instrumentation*, vol. 3, no. 08, p. S08004, 2008. [Online]. Available: http://stacks.iop.org/1748-0221/3/i=08/a=S08004

[2] I. M. de Abril and C.-E. Wulz, "The CMS Trigger Supervisor: Control and Hardware Monitoring System of the CMS Level-1 Trigger at CERN," Ph.D. dissertation, Barcelona, Autonoma U., 2008. [Online]. Available: http://cds.cern.ch/record/1446282

[3] D. Abbaneo, A. Bal, P. Bloch, O. Buchmueller, F. Cavallari, A. Dabrowski, P. de Barbaro, I. Fisk, M. Girone, F. Hartmann, J. Hauser, C. Jessop, D. Lange, F. Meijers, C. Seez, W. Smith, *The Compact Muon Solenoid Phase II Upgrade Technical Proposal*. European Organization for Nuclear Research (CERN), 2015.

[4] A. Breskin and R. Voss, *The CERN Large Hadron Collider: Accelerator and Experiments*. Geneva: CERN, 2009.

[5] World Wide Web Consortium (W3C), "Custom elements," https://w3c.github.io/webcomponents/spec/custom/, 2015.

[6] Mozilla Developer Network, "Web components," https://developer.mozilla.org/en-US/docs/Web/Web_Components, 2014.

[7] C. House, "Html5 web components: The solution to div soup?" https://www.pluralsight.com/blog/software-development/html5-web-components-overview, 2015.

[8] M. Bazon, "Uglifyjs the compressor," http://lisperator.net/uglifyjs/compress, 2012.

[9] Google Developers, "Annotating javascript for the closure compiler," https://developers.google.com/closure/compiler/docs/js-for-compiler, 2016.

[10] "@use jsdoc," http://usejsdoc.org/, 2011.

[11] A. D. T. A. Brian Birtles, Shane Stephens, "Web animations," https://w3c.github.io/web-animations/, 2016.