ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP090

WEB APPLICATION PACKAGING – DEPLOYING WEB APPLICATIONS AS TRADITIONAL DESKTOP APPLICATIONS IN CERN'S CONTROL CENTRE

M. von Hohenbühel*, S. Deghaye, E. Galatas, E. Matli, E. Roux
CERN, Geneva, Switzerland

Abstract

Web applications are becoming increasingly performant and are now capable, in many cases, of replacing traditional desktop applications. There is also a user demand for web-based applications, surely linked to their modern look & feel, their ease of access, and the overall familiarity of the users with web applications due to their pervasive nature. However, when it comes to a Controls environment, the limitations caused by the fact that web applications run inside a web browser are often seen as a major disadvantage when compared to native desktop applications. In addition, applications deployed in CERN's Control Centre (CCC) are tightly integrated with the control system and use a CERN-specific launcher and manager that does not easily integrate with web browsers. This paper presents an analysis of the approaches that have been considered for deploying web applications and integrating them with CERN's control system. The implications on the development process, the IT infrastructure, the deployment methods as well as the performance impact on the resources of the target computers are also discussed.

INTRODUCTION

Over the last decade, the use of web-applications has become increasingly prominent and CERNs Controls applications have not escaped this trend. Even if nowadays, web applications are still a relative minority in CERN's Control Centre (CCC), it is envisaged that they could represent more than 50% of the applications used in the control rooms within the next 5-10 years. The workstations deployed in CERNs control rooms are typically computers connected to two or three displays as depicted in Fig. 1, and are responsible for running several controls applications concurrently. Most of these applications require access to special data like the dynamic beam configurations orchestrated by the Controls Timing system. As such, the applications are typically controlled through a CERN-made tool, the Common Console Manager (CCM) [1].

The CCM is used to launch and manage all the applications, keeping them in the correct context and providing them with all necessary information. In a nutshell, the CCM allows the operators to run their applications, control their screen positions, and minimise/maximise them with a feature comparable to virtual desktops, depending on which accelerator beam they want to control or observe. The introduction of web-applications caused a few integration issues with the CCM, ranging from inconsistent window titles, dependency on browser support, impact on the computing resouces, and more.

One possible solution to this problem is to *transform* the web-applications into desktop applications. This paper reports on the work done to compare the industry standard called 'Electron' with a very new, but very promising framework called 'Tauri', which focuses more on resource optimisation. In addition, studies on the performance of these solutions compared with a normal web browser running the current web-applications are also reported.

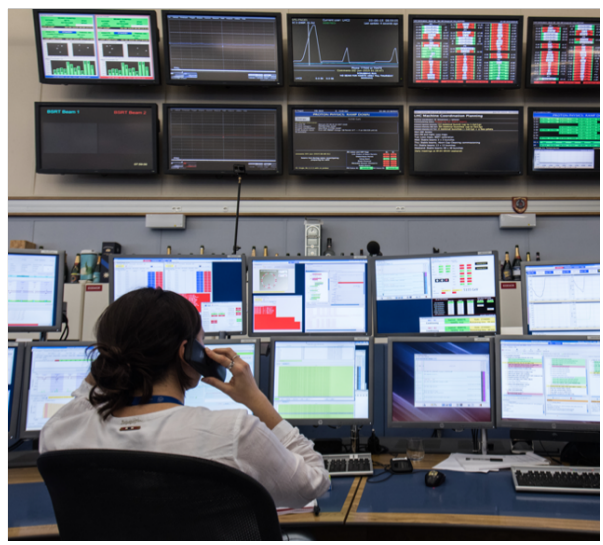

Figure 1: Workstations in the CCC.

PROOF OF CONCEPT PROJECT

A Proof of Concept (PoC) project was launched to select the best tool for web application deployment in the CERN Control Centre (CCC), and to gain a practical understanding of the impact of these solutions on the CCC Technical Console (TC) computers. As such, the PoC had 2 main objectives:

1. Create a working prototype of a representative web application, running as a desktop application (i.e. no browser), using several packaging solutions.
2. Run various benchmarks to compare the performance, and the load on the TCs, with respect to that of a conventional web browser.

* maximilian.freiherr.von.hohenbuehel@cern.ch,
stephane.deghaye@cern.ch,
epameinondas.galatas@cern.ch,
emanuele.matli@cern.ch,
eric.roux@cern.ch

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP090

Two solutions were studied, Electron [2], the de facto option, and Tauri [3]. At the time of writing, a complete evaluation of a third solution, Wails [4], was not possible due to the product maturity not yet reaching expectations. CERNs Web Rapid Application Platform application (WRAP) [5] was selected as the most relevant application to be used for the PoC work. Figure 2 shows a screenshot of a WRAP application as it was configured for the tests, with five charts connected to different data sources, asynchronously providing live updates.

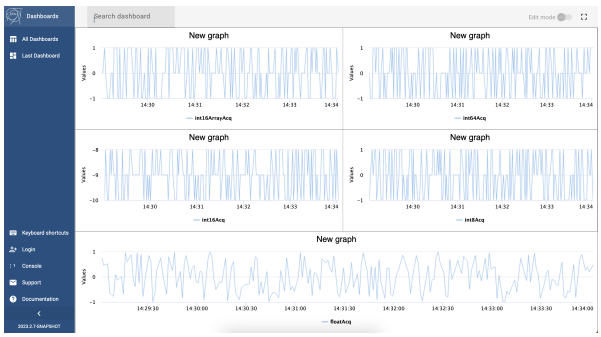

Figure 2: WRAP-based testing application (live updates).

Before presenting the benchmark results, the next section introduces the packaging frameworks, and the preparatory work needed to use them.

USING THE PACKAGING FRAMEWORKS

Electron

Electron is a mature packaging framework that exists since 2013. It is widely used in industry, offers a lot of support, and is thoroughly tested. The core concept of Electron is that it provides a Chromium web browser [6], a Node.js server [7], and the actual target web-application to be deployed, in a packaged format that can be launched from any PC.

Electron provides a lot of versatility when it comes to adding functionality. It provides the ability to intercept all requests, which allows for a way to add headers, redirect them or provide custom certificate validation (see section *Self-signed certificates*). Furthermore, it gives access to a variety of OS-level features, normally not possible from within a web browser. These range from managing windows, to having inter-process-communication (IPC), and other features like full screen. From a technology perspective, Electron uses JavaScript, which is already used at CERN for the web applications themselves. As such, Electron does not add any extra burden in terms of an additional technology.

Tauri

Tauri is a much more recent framework, with its first release being made in June 2022. As such, in terms of the PoC work, it was expected that Tauri would be less well tested and that certain issues would require CERN-specific developments in order to move forwards. Nevertheless, Tauri is a promising framework and it was considered well justified to gain practical experience. The core concept of Tauri is focused on efficiency and security. It is fully written in Rust [8] and, with the use of WebKit [9], it relies on the OS-provided browsers and shared libraries. As such, this allows Tauri to build into a tiny binary.

Table 1: Comparison of File Sizes After Packaging

Application	Size in MB	Prerequisites on target system
Wrap build	4.8	
Electron AppImage	94	FUSE
Electron Binary	240+	
Tauri AppImage	68	FUSE, WebKit
Tauri Binary	13	WebKit, glibc

The evolution plans for Tauri could raise concerns, as some factors could affect implementation in the CCC at CERN. Most notably, the Tauri developers are considering moving away from WebKit in favour of Servo [10], a fully Rust-based browser, or the Chromium web-engine. While this change should greatly improve the performance of Tauri, there have not yet been any prototypes and there is currently no prediction of when this change may happen. In the best case, this change will not affect the implementation at all, but it might require to re-develop the packaging. In addition, since Tauri is written in Rust, all interactions and plugins require the CERN development team to be capable of writing and maintaining Rust code, which is currently not used.

Packaging Formats

In general, packaging a web-application is not straightforward, since it requires some form of a web-engine on the target device. This challenge has been solved by different packaging formats.

AppImage is a format that is generally known to be able to be run on any Linux distribution [11]. It requires few pre-installed libraries on the target system; Only FUSE [12] and the correct glibc version have to be installed. Due to compatibility issues, one must install the exact or newer versions that were used at build time on the target operating system. This format seems very promising since CERN's Controls computers are running Linux.

Binary is a format that is provided by the packaging frameworks themselves. In general, it contains the precompiled code and does not include any of the required libraries or external dependencies like the AppImage format.

Table 1, shows the results of trying both formats and the corresponding final packaged application sizes. As shown, the final build of the WRAP web-application takes very little space. In contrast, any AppImage file is expected to be large, since it embeds a copy of all necessary dependencies and libraries. Compared to the Tauri AppImage, the Electron AppImage file is a lot larger, due to the fact that it includes a local copy of the Chromium browser.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP090

Although Electron cannot be compiled into a binary like Tauri, there is an option to download a pre-made Electron binary executable. This executable can then be combined with the required dependencies, consisting of Chromium, Node.js, and the target web-application, to create a deployable folder that resembles a non-compressed AppImage solution. This brings support for native options without any third-party tools but comes with the cost of a very large resulting file size. This large size and the manual build steps make this a sub-optimal solution for the targeted use-case and it was therefore not pursued further.

As mentioned earlier, Tauri is written in Rust and can be compiled into a very small binary file. However, this comes with a prerequisite to have several dependencies (WebKit and libopengl) already installed on the target system. For Red Hat Enterprise Linux 9 (RHEL9), used at CERN for Controls services, the additional dependencies are less than 300 MB in size, and only need a single copy on the host in question.

Third-party libraries exist to create an AppImage file for Electron packaged applications. In the PoC, "Electron Forge" [13] was chosen, because it allows the required configuration to be kept inside the existing web-application's "package.json" file, which avoids any additional complexity. After optimization via the configuration options available, it was possible to reduce the final AppImage size to 94MB, compared to an initial image size of about 750MB when using the default configuration. Electron Forge also allows to customize the build and output paths, which makes it easier to integrate the creation of the AppImage file into the GitLab CI/CD pipeline, used for all Controls web-application developments.

Tauri comes with integrated build configuration and bundling scripts and supports the same configuration options as Electron Forge. However, to avoid incompatibility issues with libraries and dependencies, the Tauri configuration needs to take into account the specific target systems. This means for multiple target systems, multiple Tauri configurations are needed. For CERN Controls, this is not constraining as the same version of Linux is deployed on all Controls computers.

Self-Signed Certificates

For all internal web products, CERN relies on self-signed certificates, which means that CERN acts as its own Certificate Authority (CA). The CERN-internal use of HTTPS is fully secure, but by default web browsers do not accept self-signed certificates unless they have been installed as being trusted on the local machine. As such, this requires that end-users need to either install the CERN CA certificate on their computer, or to accept an 'unsafe' connection. During the PoC implementation, this was problematic, since the Electron and Tauri frameworks rely on having working HTTPS connections with publicly-accepted certificates only.

To overcome this limitation, Electron provides a configuration flag which can be changed to accept insecure connections. This was an easy solution that worked well. For Tauri, and the CERN Controls use case, the workaround is to write two custom plugins:

1. Firstly, a plugin is needed to replace the standard HTTPS-client, with a custom version that allows insecure connections.

2. Since the above plug-in only covers the HTTP protocol, and WRAP relies heavily on WebSocket communication for receiving live device data [14], a second plugin for WebSocket Connections is required.

This approach was unappealing, since it requires extending Tauri with custom code, and having Rust knowledge to do that, and then being faced with the associated long-term maintenance. As a temporary solution, to move forward with the PoC and bench-marking, it was decided to interface the Tauri-based prototype directly to one of WRAP's HTTP server nodes, by bypassing the proxy layer providing HTTPS. This solution is not viable for a final implementation, but it makes no difference to the bench-mark results. Looking further ahead, the plans are to move all Controls services to run on Kubernetes clusters [15]. At that time, a switch from self-signed certificates to the certificate provider "Let's Encrypt" [16] is foreseen, which will properly resolve this issue for the use of the packaging frameworks.

Accessing the Backend

Web applications are usually deployed on web servers, which the client connects to, to download the frontend application, which then runs in the client's browser. Any subsequent request for data is generated from the client and directed to the same server. The base path is therefore known by the application and only relative URLs are needed to access the resources on the server. Packaged applications distribute the frontend part with the framework and run it on a self hosted web server. For this reason, it needs to be provided with the base URL in order to connect to the correct backend server. In addition, the WRAP development process uses four different environments for development, testing, provisioning and production, each hosted on separate servers. For the frameworks to successfully send the requests to the correct servers, the corresponding base URL has to be specified. This problem was solved by introducing a variable that can be configured before each build and used by an interceptor, taking care of all the networking aspects. Since the interceptor is at the frontend application level, a single implementation solves the issue for both Electron and Tauri.

BENCHMARK PROCESS

Methodology

For the benchmark, a variety of different WRAP-based applications were created to represent typical use-cases and simulate different kinds of load. The metrics were gathered

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP090

using the Linux command "top" [17]. Top provides a dynamic and interactive terminal interface that displays real-time system resource utilization, prominently highlighting CPU and memory measurements for both the overall system and individual processes. The relevant metrics for the PoC benchmarks are the memory usage, system-wide and from the tracked processes, and the load on the CPU. It is difficult to identify the exact resources that a process uses, but by looking at the Residual Set Size (RSS), the real physical memory consumption can be estimated. A script was developed to filter the monitored logs and produce CSV files, which were then used to generate the graphs and the results presented below. All measurements were conducted in a controlled environment (i.e. same conditions and same procedure) that was monitored before each run to establish the following baselines:

- The system with no processes running
- The CCM launched, but idle.
- The empty framework running.
- The framework running with WRAP's landing screen (no activity)
- The WRAP applications running.

Afterwards, the test application was launched with six instances running in parallel. Having six instances running simulates the real need of having many applications open at the same time in the CCC, knowing that all of them have to run smoothly.

Benchmark Results

Figure 3 shows memory usage for a series of tests performed on different web engines, ranging from web browsers (Chromium and Firefox) to the packaging solutions (Electron, Tauri AppImage, and Tauri Binary). It was clear that Chromium was the best browser option in all tests. It stays stable and only the addition of the WRAP charts, which require a lot of rendering activity, increases the resource consumption.

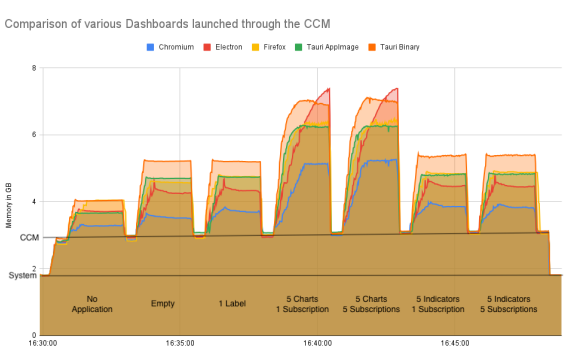

Figure 3: Benchmarks across various browsers.

Electron was generally stable with relatively low low memory usage. However, the charts were somewhat poorly handled, and when it comes to running multiple instances, Electron showed that it is resource-hungry, as few resources are shared between distinct Electron instances.

For Tauri, the generated Binary performed significantly worse than the respective AppImage. This was surprising and even the Tauri developers had no obvious explanation. The only reasonable explanation so far, is that the dependencies injected inside the AppImage might have better efficiency, since they are based on a different version, built on a different system.

Looking in more detail at Tauri, as shown in Fig. 4, the networking aspects do not consume much memory. The two main bottlenecks are the parsing of the JSON values into objects while passing them to the charts and the rendering needed to show the live data. Both of which are handled by the 'WebKit Web process'.

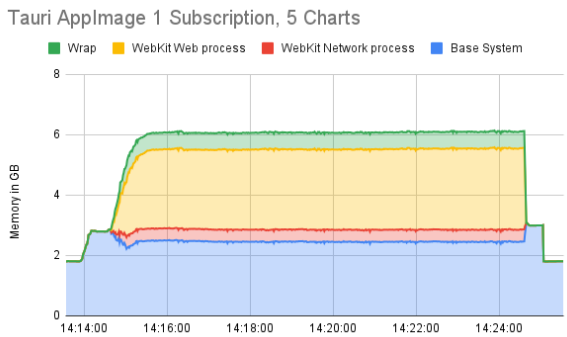

Figure 4: Tauri memory usage by main processes.

The CPU usage metrics gave interesting insights into the rendering and data (de)serialization aspects. It is noticeable that the Chromium-based applications (Chromium, Electron) have a higher CPU usage than the others.

Looking at Fig. 5, it is clear that the number of subscriptions has little impact on CPU, otherwise there would be a constant CPU usage, meaning that the CPU spikes are mainly due to the rendering.

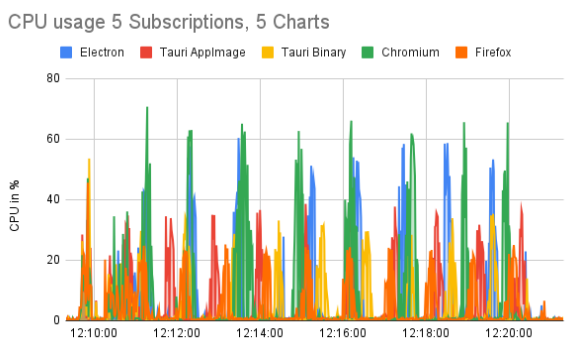

Figure 5: Total CPU usage for 5 subscriptions.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP090

Figure 6 shows CPU usage for each framework whilst displaying live heatmaps (i.e. 2D-array data) that were fully initialised with 10'000 values. As shown, this is CPU-intensive and there are clear variations in load for the different packaging solutions and browsers.

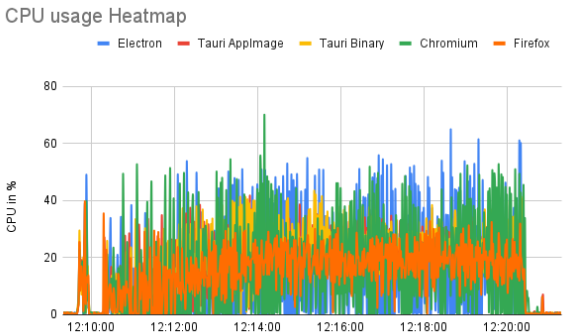

Figure 6: WRAP CPU usage when displaying heatmaps.

Surprisingly, when looking at the memory usages in (Figs. 7 and 8), it seems that Tauri does not handle heatmaps as well as the other solutions. The other solutions also use less memory for the heatmaps than they do for the regular charts, even though the heatmaps have a lot more values behind them. In contrast, Tauri used less resources for the regular charts. Overall, this gives a good example of how the different web-engines handle different data structures differently from a resource perspective.

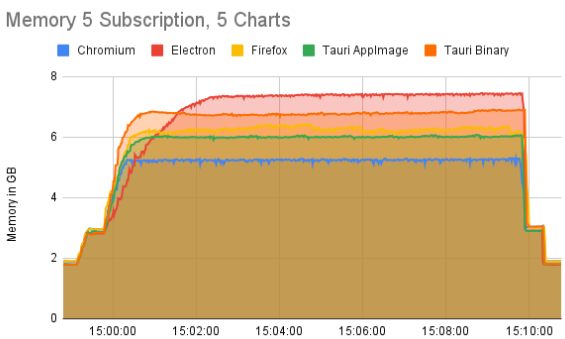

Figure 7: Memory usage for 5 subscriptions.

As a side note, it was also observed that the start-up time of Tauri is approximately twice as fast as the web browsers, and 3.75 times faster than Electron. This can contribute to a better user experience (UX), particularly in the CCC where applications maybe started and closed many times per day.

NEXT STEPS AND FUTURE OUTLOOK

After the benchmarks, several questions remain. On the framework side, some open issues need to be addressed to provide 100% usability inside the CCC. For example, the integration of command-line arguments to control WRAP is not yet ready. For Tauri, the need for plugins to be developed and the associated maintenance has not yet been pursued.

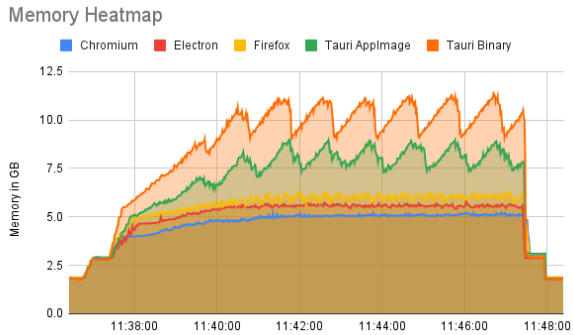

Figure 8: Memory usage with WRAP displaying heatmaps.

This raises the question: is Tauri sufficiently better than Electron, in terms of performance, including start-up time, and resource consumption to justify the addition of Rust in the Controls technology stack? At the same time, would it be wise to use Tauri as it is at the time of writing, knowing a significant change in its backend implementation could happen in the near future? Such a change will most probably have beneficial consequences in terms of performance and ease of deployment, however any integration work referenced in this paper may need to be redone completely. For these reasons, it was decided to proceed with Electron for CERN's Controls web-applications, for the foreseeable future. At the same time, a close eye will be kept on Tauri, and it is likely that the situation will be re-evaluated once the Tauri web engine has been changed and when a production-ready Kubernetes platform is available for CERN's accelerator controls services.

SUMMARY

The packaging PoC was a clear success, resulting in an Electron-based deployment of a WRAP web-application, that is being used on a daily basis, from CERN's Control Centre, running smoothly alongside a plethora of Java Swing and PyQt applications.

The PoC gave important insights into the use of packaging solutions. It also revealed that for now, Electron is the only viable solution for CERN, since it does not require any custom extensions to connect to the backends and does not depend on any additional installations on the CCC Linux consoles.

Compared to Electron, Tauri provides a smaller binary size, faster start-up, and a lower CPU usage in several scenarios. In addition, while Chromium is the fastest and most stable web-engine, Tauri managed to stay on top for several benchmarks, clearly showing its potential. The foreseen evolution of Tauri to integrate Chromium or Servo is appealing, while the introduction of Kubernetes clusters for CERN's Controls will help address other issues such as certificates, removing the need to develop specific plugins. Electron wins, for now.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUPDP090

REFERENCES

[1] CCM (CERNs Common Console Manager), https://be-dep-css.web.cern.ch/node/123

[2] Electron, https://www.electronjs.org

[3] Tauri, https://tauri.app

[4] Wails, https://wails.io

[5] E. Galatas *et al.*, "WRAP - A Web-based Rapid Application Development Framework for CERN's Controls Infrastructure", in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 894–898. doi:10.18429/JACoW-ICALEPCS2021-THPV013

[6] Chromium, https://www.chromium.org/Home/

[7] Node.js, https://nodejs.org

[8] Rust, https://www.rust-lang.org

[9] WebKit, https://webkit.org

[10] Servo, https://servo.org

[11] AppImage, https://appimage.org

[12] FUSE, https://www.kernel.org/doc/html/next/filesystems/fuse.html

[13] Electron Forge, https://www.electronforge.io

[14] E. Galatas *et al.*, "Improving CERN's Web-based Rapid Application Platform", presented at ICALEPCS'23, Capetown, South Africa, Oct. 2023, paper TUPDP089, this conference.

[15] T. Oulevey *et al.*, "An Update on the CERN Journey from Bare-Metal to Orchestrated Containerization for Controls", presented at ICALEPCS'23, Capetown, South Africa, Oct. 2023, paper TH2AO03, this conference.

[16] Let's Encrypt, https://letsencrypt.org

[17] Top, https://man7.org/linux/man-pages/man1/top.1.html