

Adopting WebAssembly when Migrating .NET Applications to the Web ^{*}

Leonardo Estrela^{1,2}, Paulo Maio¹, and José António Silva²

¹ Instituto Superior de Engenharia do Porto {1171430,pam}@isep.ipp.pt

² DevScope

{leonardo.estrela,joseantonio.silva}@devscope.net

Abstract. This work assess the pertinence and validity of using WebAssembly, a recent W3C recommendation, to migrate from Microsoft .NET Windows Forms desktop applications to web/mobile applications. For that, a long established document data capture application, called SmartDocumentor, developed by DevScope is taken as a case study. A proof-of-concept for the Review Station module has been devised using Blazor and evaluated through experiments. The achieved results shows that the proof-of-concept application is able to match the desktop application in most of the considered criteria, even though there is still room for improvements (e.g. file parsing time). Thus, when it comes to migrating .NET applications to the Web, (Blazor) WebAssembly despite being a young and immature technology, is already a solid alternative for the migration of .NET applications.

Keywords: WebAssembly, Web development, App Migration, Blazor, Uno Platform

1 Introduction

Throughout the last decades, software companies have successfully developed and delivered desktop applications backed by Microsoft's .NET Framework and Windows Forms library. However, advantages of the ever increasing internet use and accessibility, is leading companies to shift their development from desktop applications to web and mobile applications. While this shift is straightforward for new applications, for established ones it requires a (complex) migration process encompassing the full application or just a few modules. This either implies the development team to adopt, at least partially, a new technological stack comprehending, for instance, HTML, CSS and Javascript frameworks whose proficiency on it should be already owned or be acquired by the team. Yet, migration processes commonly have to cope with requirements/concerns such as (i) to reuse as much code as possible from the original application, (ii) to replicate the same user experience, (iii) to mitigate the constant need of internet connection, (iv) to keep high performance standards and (v) the ability to render the same application on different environments like desktop, web and mobile.

^{*} Supported by DevScope.

In this context, WebAssembly³, recently endorsed by W3C, is seen as an interesting option to this kind of scenarios, considering that it allows for offline apps and it performs considerably better than JavaScript on certain situations [1]. Thus, through a case study, this work aims to assess the pertinence and validity of using WebAssembly in the migration process of .Net Windows Forms applications to web/mobile applications.

The rest of paper describes the case study (section 2) which is followed by a technological context analysis (section 3). Further, it is described the carried out proof-of-concept and its evaluation process (section 4). Finally, some conclusions are drawn in section 5.

2 Case Study

DevScope's SmartDocumentor⁴ is a software solution tailored to reduce companies manual labor and operational costs upcoming from extracting, analyzing and processing data of paper' documents such as invoices and receipts. It comprehends four main processes:

- Scanning Process, whose purpose is getting a digital copy the paper' documents that need to be processed. Each scanned document is, thus, stored and treated as a task on further steps.
- Processing Process, whose purpose is automatically extracting data from, previously acquired, digital documents by employing techniques/engines for printed and hand-printed text recognition (OCR and ICR), brand recognition (OMR) and barcode reading. Processed documents are stored in a PDF Double Layer format that allows adding an invisible layer of search-able text document, while maintaining the initial appearance of the document.
- Review Process, whose purpose is to enhance the system with the ability of getting a human confirmation that the extracted data has the required quality/value. Thus, the document' extracted data/values is presented to the user in a way that it is visually possible to him/her validate such data in terms of integrity and accuracy. Figure 1 depicts a screenshot of this process being executed into Review Station component of SmartDocumentor.
- Integrating Process, whose purpose is automatically make the extracted data available and useful for (other) information and managements systems that are being used into the organization such as ERP and CRM systems.

This work focuses only on the Review Station. Currently, it is delivered to users as a desktop application written in Microsoft Windows Forms .NET library. However, to meet new customers and business requirements, DevScope intends to evolve/migrate this tool in a way that it becomes possible to deliver it either as a multi-platform web application. There is a major concern with avoiding/minimizing (i) changes regarding users' interaction and usability; and

³ <https://www.w3.org/TR/wasm-core-1/>

⁴ <https://www.smartdocumentor.net>

(ii) the bulk of the application to be rewritten. On the other hand, it has been noticed there are already some frameworks claiming the ability to convert .NET code into WebAssembly code such that it can be parsed by a common browser.

- Emulate the basic functions of SmartDocumentor's Review Station:
 1. Display a zoom-able, drag-able image;
 2. Allow the user to navigate through image, using image coordinates, using a button that signals the image viewer to zoom and center at a certain point of the image;
 3. Read XML and JSON files composed of information regarding multiple locations of the image (coordinates and other information);
- Reuse as much code from the original desktop app as possible;
- Study the possibility of having a single codebase, but having the ability to render the application to multiple target environments (e.g. desktop, web, mobile);
- Provide an overview of the advantages and disadvantages of using a WebAssembly based frameworks for web-development.

This section presents a brief technological overview by (i) pointing out some differences and similarities between WebAssembly and JavaScript (section 3.1); and (ii) analysing and comparing two WebAssembly frameworks available on the market (section 3.2)).

According to [7], WebAssembly is a safe, portable, low-level code format designed for efficient execution and compact representation. Its main goal is to enable

high performance applications on web browsers, without making any specific assumptions or providing features that are exclusive to the Web which, ultimately, allows it to be employed in many different environments.

Since WebAssembly provides a way to run web client-side code, written in multiple languages (traditionally not used for that purpose), at near native speed, it might have a huge impact on the web platform development. This feature allows, for instances, one to develop a client-side part of a web application without using JavaScript, and using instead Microsoft C#. Despite this, it is important to stress that WebAssembly is not a replacement of JavaScript. Both can, and should, coexist. JavaScript is powerful enough to solve most of the problems one have when building a a client-side web application. But, it has also showed that it has some limitations on more complex and intensive scenarios as, for instance, 3D game, video editing, and other domains that require native performance [7].

Different aspects contribute to WebAssembly performance being superior to JavaScript [1], namely:

- It is more compact than JavaScript, which makes fetching faster;
- It is much closer to the machine than JavaScript such that its compiler does not have to spend time running the code to observe which types are being used before it starts compiling optimized code, or compile different versions of the same code based on those observed different types. Thus, since the compiler does not need to make assumptions regarding the types being used, re-optimization cycles are useless.

Hence, many of the optimizations used by developers to make JavaScript more efficient are not necessary on WebAssembly, which tends to be faster than JavaScript.

3.2 WebAssembly Frameworks

As of the time of writing this work, two multi-platform WebAssembly frameworks were identified: (i) Blazor ⁵ and (ii) Uno Platform ⁶.

Blazor is a framework developed by Microsoft that allows for the development of multi-platform .NET apps, using reusable components written with .NET and Razor [2]. These components, which are elements of the UI, such as an entire page or just a section of another component, are written in the form of a Razor markup page. They define the UI rendering logic, handle user events, and most importantly, can be nested and reused. Currently, Blazor has two distinct models that can be used by developers: (i) Blazor Server and (ii) Blazor WebAssembly. The Electron Renderer and the Mobile Blazor Bindings Renderer are still in experimental phase, and are not stable enough to be used on a production environment.

⁵ <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>

⁶ <https://platform.uno>

Blazor WebAssembly is the main hosting model for Blazor. With this hosting model, the Blazor app, all its dependencies and the .NET runtime are downloaded to the browser, and executed there through to WebAssembly [4].

Uno Platform is an open-source, graphical user interface that allows for apps based on UWP [5] to run on Mobile, Desktop and Web environments [9]. The code is written using the UWP tools made available by Visual Studio, including C# and XAML, and can be run on iOS, Android and inside a browser through WebAssembly. On iOS and Android, Uno Platform relies on the Xamarin Native stack ⁷. As so, Uno Platform provides the ability to run a single codebase, C# and XAML apps on all these platforms.

Just like Blazor WebAssembly, Uno Platform's web head relies on Mono [6] to work. Also, both allows for the creation of Progressive Web Apps (PWA). According to [11], a PWA provides an app-like experience on desktop and mobile, and it is delivered directly via the web. They are, in short, fast and reliable web apps, that mimic the behaviour of native (mobile) apps. They are designed to be capable, reliable and install-able, and to mimic the best that web apps and native apps have to offer. This is an interesting feature that should be considered, when choosing the WebAssembly framework.

Table 1 presents a comparison between the Blazor WebAssembly and the Uno Platform. Observing this table, it becomes clear that Blazor fits the relevant requirements almost perfectly, while Uno Platform would require a few compromises. Although Uno Platform offers more renderers, as of the time of this review, Blazor will soon have the same renderers. Plus, only the web renderer is important for the PoC application. As such, Blazor WebAssembly is chosen as the framework on which one will develop our proof-of-concept application.

Table 1. Comparison between Blazor WebAssembly and Uno Platform.

| Requirement/Platform | Blazor | Uno Platform |
|--------------------------------------|----------|-------------------------|
| Supports WebAssembly | Yes | Yes |
| Supports PWA | Yes | Yes |
| Supports JavaScript Interoperability | Yes | Limited |
| Suitable C# Image Viewer | No | No |
| Supports OpenSeadragon | Yes | No |
| UI Language | HTML/CSS | XAML |
| Available Renderers | Web | Web, Mobile and Desktop |

4 Proof-of-concept

This section summarizes major decisions and compromises made during PoC development (section 4.1) and, further, describes the carried out PoC' evaluation process (section 4.2).

⁷ <https://dotnet.microsoft.com/apps/xamarin>

4.1 Solution Development

A major decision concerning the PoC development relates with the adoption of an image viewer capable of meeting previously stated requirements (cf. section 2). Therefore, a study to identify and select an image viewer was conducted. First, this study focus on C#.Net target image viewers (e.g. Syncfusion's WPF PDF viewer⁸, PDFTron's WebViewer⁹), but none proved to be ideal for the problem at hands as reported in Table 1. Considering this outcome, one turn the study to Javascript target image viewers. In this regard, the OpenSeaDragon viewer [8] come across as fitting (almost) perfectly to the PoC requirements. Its adoption was possible due to the JavaScript interoperability support provided by Blazor.

Due time and effort constraints, it would not be possible to replicate the exact flow of the original application, in which the Review Station requests tasks (i.e. processed image files) that have already been parsed into domain objects by other components. Instead, PoC application works isolated making that the parsing process occurs every time the user tries to access a task.

At the end, one can say that the PoC application developed during this work performs and fulfills all the requirements and functionalities previously stated. A screenshot of the application's UI is depicted in Figure 2. This application has been tested with integration and unit tests, and functions as a PWA.

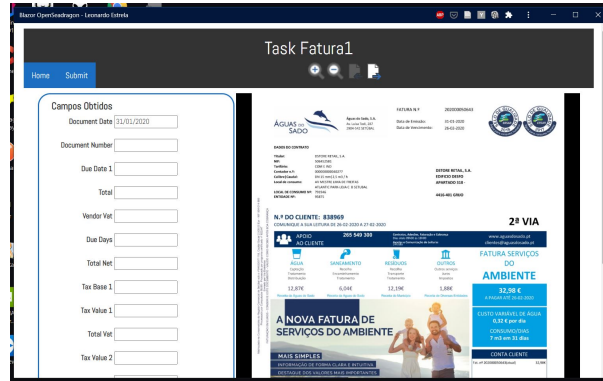


Fig. 2. UI of PoC Review Station application.

4.2 Solution Evaluation

To evaluate the developed solution, several time measures were taken, in order to compare the performance of the original application and the Blazor proof-of-concept. For this, three different tasks were defined as follows:

⁸ <https://www.syncfusion.com>

⁹ <https://www.pdftron.com/webviewer/>

1. Task composed of just 1 page, with a simple text phrase written on it;
2. Task composed of just 1 page, but with numerous Areas identified by the OCR engine;
3. Task composed of 2 pages, and with numerous Areas identified by the OCR engine.

All tasks were reproduced on (i) the original SmartDocumentor Review Station and (i) the PoC Review Station developed during this work. Obtained results are presented in Table 2.

Table 2. Time results of evaluation task 1, 2 and 3 in seconds. "Inst." is used as an abbreviation of instantly.

| Functionalities | Task 1 | | Task 2 | | Task 3 | |
|--|---------|----------|---------|----------|---------|----------|
| | Blazor | Original | Blazor | Original | Blazor | Original |
| Zoom and pan on image | Inst. | Inst. | Inst. | Inst. | Inst. | Inst. |
| Change Page | Inst. | Inst. | Inst. | Inst. | Inst. | Inst. |
| Set value to Textbox (QuadTree) | Inst. | Inst. | Inst. | Inst. | Inst. | Inst. |
| Setup (without parsing) | <1 sec. | 3 sec. | <1 sec. | 3 sec. | <1 sec. | 3 sec. |
| Setup (with parsing) | <1 sec. | 3 sec. | 10 sec. | 3 sec. | 20 sec. | 3 sec. |

Analysing the results, it is possible to see that the solution is mostly successful in replicating most of the functionalities offered by the original Review Station, particularly regarding the core functions of the module, such as zooming, panning, changing pages and attributing values (by querying the QuadTree structure) on a Task.

Respecting the setup process, however, it is not possible to draw a solid conclusion regarding the performance of the proof-of-concept. Although, the PoC application performs a lot worse than its desktop counter part. In this respect, it is pretty clear that the main bottleneck for the Blazor Review Station is the time needed to parse large, complex files into objects. This is mainly due to Blazor (i) lacking multi-threading support and (ii) having a memory consumption limit, which is severely debilitating to the performance of a program, in situations such as the ones presented on this project, where extremely large files must be parsed into classes. These Blazor' limitations are already being addressed by Microsoft Blazor's development team [3], who intends to present solutions as of the time of the release of .NET 5 [10].

It is important to note that the Blazor PoC takes between 1 and 2 seconds to load each component. This is tolerable, but still considerable longer than its JavaScript competitors. This is mostly attributable to the download size of the project, which is around 6 MB which, admittedly, is much more than most JavaScript apps.

Overall, while Blazor WebAssembly does perform badly when put on a situation such as this, this situation does not represent a production scenario for SmartDocumentor's Review Station since the parsing process is responsibility of

other application modules as stated on 4.1. In short, it is fair to consider Blazor WebAssembly as a possible and viable migration environment for SmartDocumentor’s Review Station, even though the many limitations of the framework should be taken into account.

5 Conclusions

After a theoretical overview of WebAssembly and Blazor, and the evaluation of a proof-of-concept solution, it is showed that Blazor WebAssembly offers a valid alternative to JavaScript frameworks as a possible environment for SmartDocumentor’s Review Station, even though the technology still presenting some concerning issues. As far as WebAssembly is concerned, it is very clear that it represents a tremendous technological achievement, that will no doubt change web programming, over the next few years.

References

1. Clark, L.: What makes webassembly fast? url = <https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/> (2017), last checked on: March 12, 2020
2. Garcia, I.: Blazor: Introduction. url = <https://www.nativoplus.studio/blog/blazor-introduction/> (2018), last checked on: May 27, 2020
3. GitHub: [blazor webassembly] serious performance issues. url = <https://docs.microsoft.com/pt-pt/aspnet/core/blazor/?view=aspnetcore-3.1> (2020), last checked on: April 28, 2020
4. Microsoft: Introduction to asp.net core blazor. <https://docs.microsoft.com/pt-pt/aspnet/core/blazor/?view=aspnetcore-3.1> (2020), last checked on: April 28, 2020
5. Microsoft: Uwp. url = <https://docs.microsoft.com/en-us/windows/apps/desktop/choose-your-platform#uwp> (2020), last checked on: April 28, 2020
6. Mono: About mono. url = <https://www.mono-project.com/docs/about-mono/> (2020), last checked on: June 5, 2020
7. Mozilla: Webassembly. url = <https://developer.mozilla.org/pt-PT/docs/WebAssembly> (2020), last checked on: April 28, 2020
8. OpenSeadragon: Openseadragon. url = <https://openseadragon.github.io/> (2020), last checked on: April 21, 2020
9. Platform, U.: How it works - uno platform. url = <https://platform.uno/how-it-works/> (2020), last checked on: June 23, 2020
10. Ramel, D.: Next for blazor: Aot for ‘massive speed gains’. url = <https://visualstudiomagazine.com/articles/2020/05/22/blazor-future.aspx> (2020), last checked on: June 23, 2020
11. Sam Richard, P.L.: What are progressive web apps? url = <https://web.dev/what-are-pwas/> (2020), last checked on: June 24, 2020