OULUN AMMATTIKORKEAKOULU

Bars Margetsch

MODERN HYBRID WEB-BASED UI FOR LEGACY AND NATIVE WINDOWS APPLICATIONS

MODERN HYBRID WEB-BASED UI FOR LEGACY AND NATIVE WINDOWS APPLICATIONS

Bars Margetsch
Bachelor's Thesis
Spring 2024
Degree Programme in Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology

Author(s): Bars Margetsch
Title of the thesis: Modern hybrid web-based UI for legacy and native Windows applications
Thesis examiner(s): Lasse Haverinen
Term and year of thesis completion: Spring 2024 Pages: 46

An ever-increasing number of Windows applications reach the end of their original UI framework's lifespan. A possible solution presented in this thesis is migration to a hybrid architecture, replacing the ageing native framework with a new, web-based user interface.

This is described through the example of a prototype application developed for Keysight Technologies' Nemo Outdoor application, which was built to determine the viability of web-based user interfaces when compared to the pre-existing native ones and identify and describe the main challenges associated with it. Considering Nemo Outdoor's highly dynamic and data heavy interface, it was the perfect candidate for evaluating the performance and development challenges of a hybrid application in an extreme case.

The prototype consists of an Angular web component, running in a Windows Forms application, which uses the Microsoft Edge WebView2 runtime to host it. The communication between it and the Nemo Outdoor backed is done through an integrated REST API.

The thesis concludes that migration to a hybrid web-based interface is viable, and while there are some initial challenges, the prototype demonstrated that it could perform in various data and processing heavy environments while still providing a responsive user experience.

Keywords: C#, .NET, JavaScript, TypeScript, React, Angular, WebView2, Hybrid application

CONTENTS

1 INTRODUCTION

The purpose of this thesis is to investigate and demonstrate the viability, as well as describe the process and challenges of migrating a traditional Windows application from a legacy end-of-life user interface framework to a modern web-based one. This process is shown and described through an exploratory prototype built for this purpose for Keysight Technology's Nemo Outdoor product. However, this can also be applied from the start for new projects as well.

Web frameworks and applications have become more mainstream, as focus increasingly shifts towards online services. While typically these operate only within the limited contexts of a standard web browser, hybrid applications offer a way to utilise these powerful web frameworks on the desktop to provide native applications with rich user interfaces. This comes with several advantages and disadvantages as well: while it is considerably easier and faster to develop with web frameworks thanks to their streamlined design and HTML and CSS being simple, this comes at the cost of the lower performance due to browser's restrictive, sandboxed architecture, and JavaScript's original design principles drastically differing from today's usage.

Evaluation of developing a hybrid application in the context of these trade-offs forms the basis of this project, and thus this thesis.

Though this process is unique to every project and there is no one-size-fits-all solution, the goal here is to describe it in more generic steps and provide insight into important considerations, through examples that can then be used as the basis for other projects.

Although an integral part of this project (about ~50%) was developing a completely new user interface and redesigning the way the application is used from the ground up, the focus of this thesis is the technical implementation. This is mainly due to the UI design work being highly unique to each product, influenced by the brand, company design standards, and – in the case of an upgrade – already established UI conventions and language.

The broader intent with this thesis is to be part of conversations within development teams by giving a rough idea to developers currently considering building a hybrid application – or simply upgrading from an end-of-life UI framework in an already established application – about what to expect.

1.1 Nemo Outdoor

Nemo Outdoor is a portable drive test tool for wireless network testing developed by Keysight Technologies, designed to test and validate device and 2G to 5G network performance.

The primary use case is mobile network performance and coverage testing by network operators: multiple consumer mobile devices (or network scanners) can be connected to a laptop at the same time, each recording network information and performing industry standard tests while simulating typical user usage (making voice and video calls, video streaming or social media app usage, etc and analysing their performance and quality). Devices used and measurements are handled and recorded by Nemo Outdoor where they can be configured and inspected live or later via measurement playback.

These measurements are typically performed during drive tests, where a device running Nemo Outdoor with several smartphones connected is driven around in a predetermined area to capture real world network data. This lets network operators find and diagnose issues with their networks and capture real world data to understand the performance and quality of their network when used by their customers. For example, this can be used to locate faulty transmitters, or particularly busy areas (e.g.: shopping centres) where the network may be overloaded during peak hours, degrading performance and quality of service along with user experience.

All this is done through Outdoor's robust data display system, which allows users to query hundreds of metrics and display them dynamically in highly customisable data windows and graphs. This extremely data heavy use-case made Outdoor the perfect proving ground a hybrid prototype.

While extremely powerful and its services invaluable, Outdoor's UI has aged considerably since the start of its development more than 20 years ago. It was built with the now no longer actively supported Microsoft Foundation Class library, an old Windows specific object-oriented library introduced by Microsoft in 1992.

The framework reaching its end of life and the application's aging user experience made it worthwhile to explore the possibility of migrating the UI to a new and modern framework. As such, along with this technical switch, a complete overhaul of both the user interface and experience was part of this exploratory prototype.

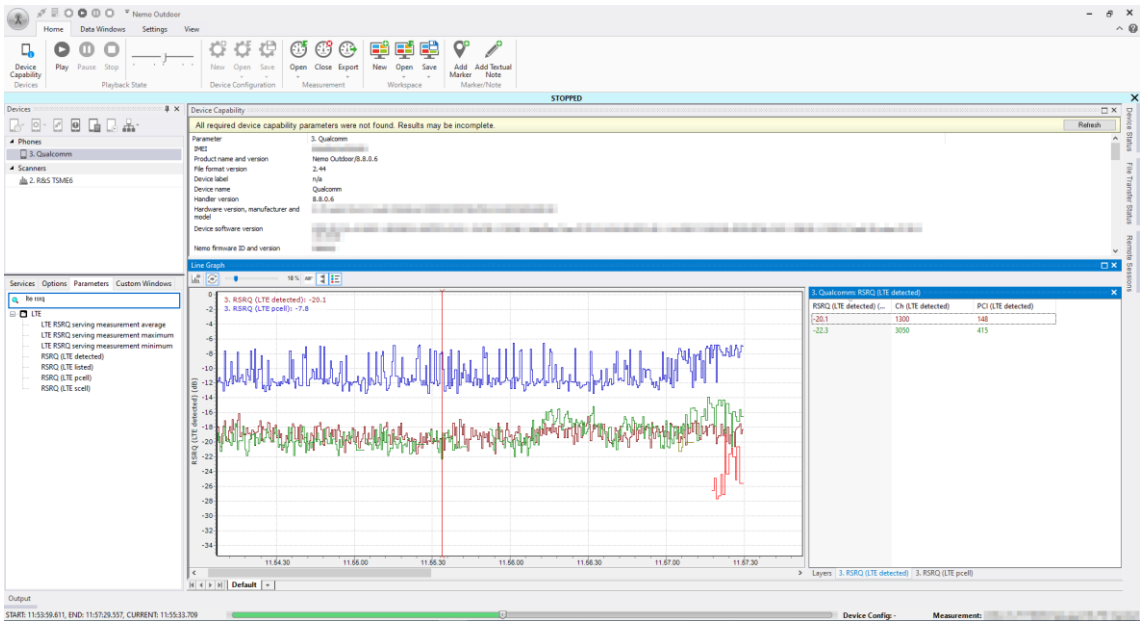

Figure 1. Nemo Outdoor's current user interface. Devices can be seen on the top left section, metric selection on the bottom left, and the measurement data on the rest of the screen.

1.2 Native, hybrid, and web applications

To understand the context of this thesis and the discussed problem, it is important to first talk about the differences between the three main application types. Modern applications can be classified into three groups: native, hybrid, and web applications. Each of these have different advantages and excel in certain scenarios, while struggle in others.

Native applications are what people typically associate with “applications”. These make up the largest group, with nearly everything installed on a Windows machine being a native application (although as this thesis describes it is slowly changing for certain use cases). They are developed with technologies and programming languages directly supported by the target environment and have access to the system and device’s resources. This means they can perform complex tasks that often would not be possible otherwise and are generally the most performant since they run closest to the operating system. These however have the issue of being costly to develop, as to utilise native system resources they must be developed per platform, which may differ substantially in architecture.

The examples here are myriad: from what this project was developed with – Visual Studio – to the word processor – Microsoft Word – this thesis was written and edited in.

Web applications on the other hand sit on the opposite side of the spectrum. They can be accessed via any modern web browser regardless of platform and can have more complex user interfaces and several robust frameworks to choose from and utilise. However, this comes at the cost of both performance and lack of access to system resources, as web browsers are sandboxed and only expose limited functionality. While easier and faster to develop, they lack the power to perform intensive tasks, and rely on backing servers instead, which means most of the time an internet connection is required to access them.
The examples here include virtually every website providing complex services, such as Google Docs.

Between the two ends of this spectrum are hybrid applications. These offer the best of both worlds: the ease of development of web applications and their robust support, combined with the ability to access system resources and perform intensive tasks. They are also cheaper to develop, as large parts of the application can be reused between platforms, drastically decreasing development times. There are still however, potential drawbacks mainly surrounding performance as later discussed, but these applications offer a good compromise between performance and versatility.
A recent example – and partial inspiration for the project – is Microsoft Teams, which was recently updated to a new hybrid architecture [1].

Developers should carefully consider these options and choose the right one for their project. While hybrid applications can be especially tempting, in many scenarios it is not worth the added complexity and overhead when only a simple user interface is required. In these cases, using a traditional native framework is probably better.

2 DESIGN AND ARCHITECTURE

2.1 Design

The overall design and requirements of the application was heavily influenced from the beginning by the rather unique use case of Nemo Outdoor: drive tests. These tests are typically done with several smartphones connected to the laptop running Outdoor, which can be cumbersome to manage in a car: with enough devices sub-controllers may be needed which further complicates cable management and placement.

A precursor product utilised an autonomous controller computer in a pelican case placed into the trunk of the car, away from the cabin, which restricted access during drive tests. This was somewhat solved by the controller hosting its own network and exposing a web interface for basic configuration for quick troubleshooting, but more advanced diagnostics or live measurement updates were not possible this way.

Although discontinued, this precursor project served as the foundation for this prototype: the basic web interface was backed by a powerful REST API not currently used by any other product. This provided the prototype with an already in-place interface and lessened development significantly on Outdoor's side.

The REST API also proved to be the perfect solution to the access problem. Currently, the UI and Outdoor are loosely coupled (although architecturally separated, they share processes, but it is possible to start Outdoor without the UI) so access in these scenarios is normally not possible, or only via remote controlling the machine itself, for example with Windows Remote Desktop. Utilising the REST API on the other hand opens the path for the UI application to remotely communicate with the measurement unit within the same network.

Considering this, the following use case scenarios were identified for the project:

- UI application and Outdoor installed on the same system: In this scenario, the application would directly control the local Outdoor instance, and would allow remote control of other

units within the same network via a different shortcut (launching the same executable, but with different launch arguments).

- UI application installed without Outdoor: In this scenario, the application would act as a remote management software to control other units within the same network.
- Only Outdoor installed: This scenario would mainly occur during drive testing, where the device running Outdoor is physically inaccessible (for example if it is in the vehicle's trunk). Connecting to the wireless network created by controller device would allow a test engineer to connect to the unit via a browser and have access to essentially the same interface as with the hybrid application, but as a regular web application, although with some limitations. This mode was intended to facilitate on-the-fly diagnostics and configuration without specialised equipment (no pre-installed software required) even by untrained personnel.

During this thesis only the first option is discussed, as the remaining two were not fully developed or would be possible without the first.

Another large part of the project was a complete redesign of the current UI and user workflow. Briefly, along with modernising the interface, the goal here was to create separate tabs or pages with different responsibilities, in a way that also acts as a sort of wizard, guiding the user through distinct steps while setting up a measurement (the steps being represented by the menu options on the sidebar, top to bottom).

An example of this can be seen with the Devices (Figure 2) and Monitoring (Figure 3) pages. These have well defined responsibilities that are immediately obvious even without necessarily understanding the application flow or configuration steps.

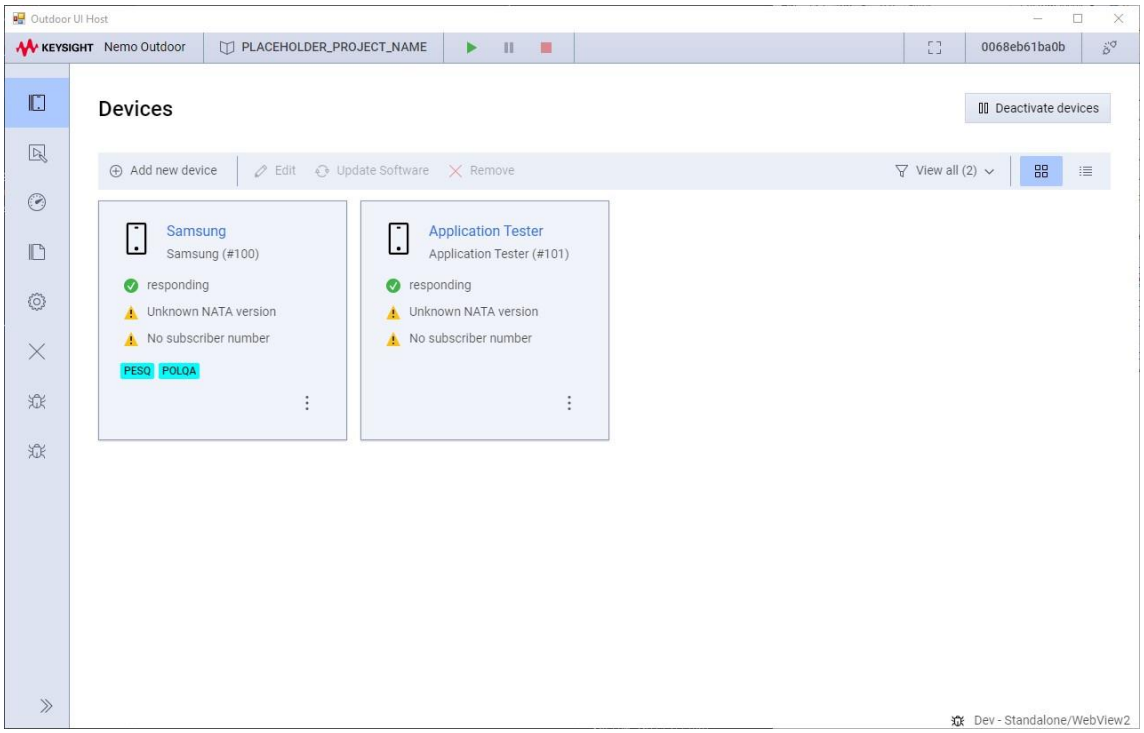

Figure 2. Devices page open, showcasing interactive cards representing connected measurement devices.

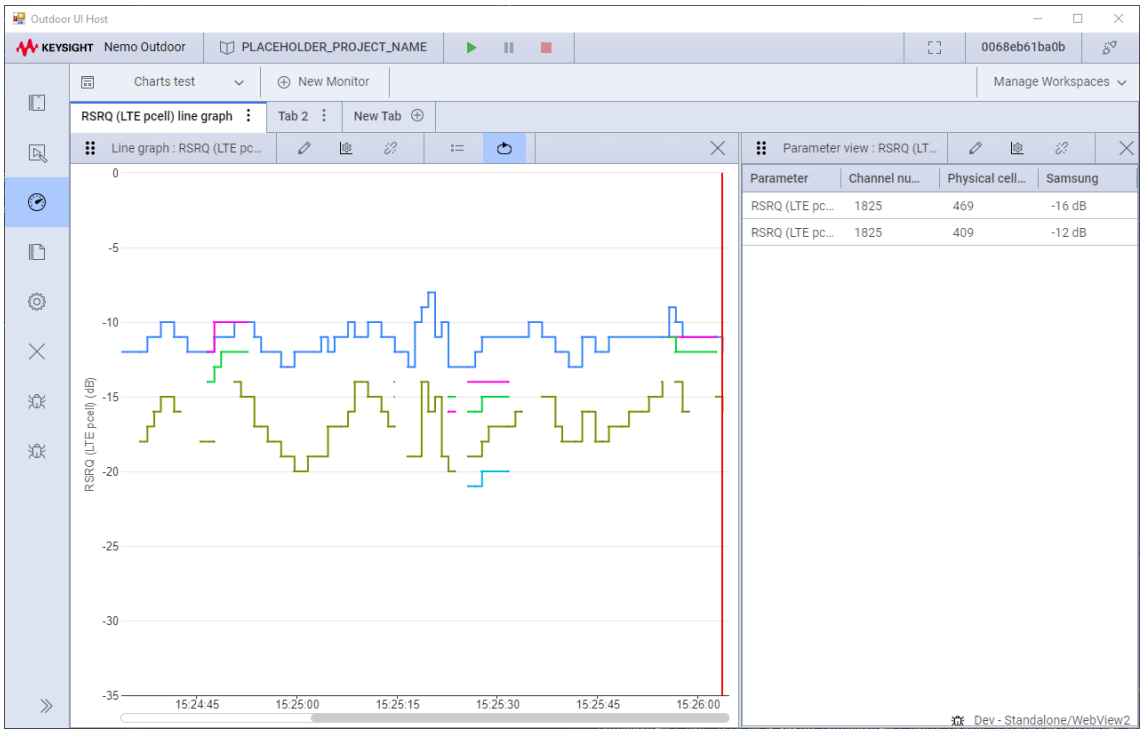

Figure 3. Example Workspace open in Monitoring page, showcasing two data windows side by side in a docked layout.

In contrast to this, the current UI of Outdoor presents these sections divided into group boxes, visible on the same screen, as shown previously on Figure 1.

2.2 Architecture

Following these use cases and design principles, a three-part architecture was designed for the project, as demonstrated by Figure 4. Since Outdoor's current UI is already structurally separate from the business logic, the decision was made to continue enforcing separation of concerns and develop the new hybrid host as a separate application.

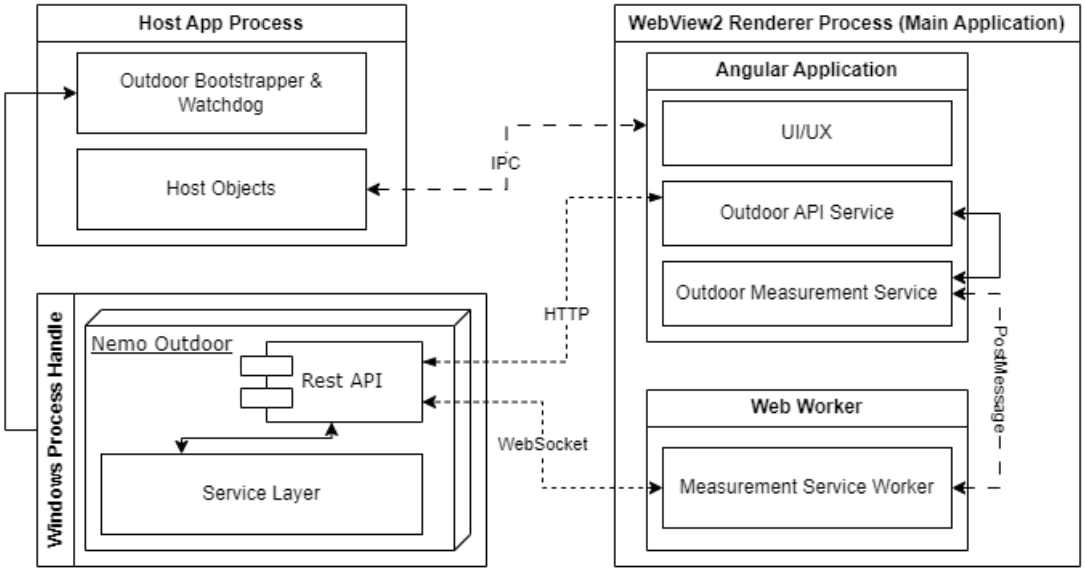

Figure 4. Application architecture diagram.

Important to note however, were these requirements not in place, it would have been trivial to instead couple the two applications more tightly, and simply replace the already separated legacy UI with a WebView2 control, directly exposing the service layer's methods to the web application within the same executable.

In this architecture the host becomes the new entry point and sits between Outdoor and the web application. It handles two core tasks: providing the web application with access to native resources, and starting and managing Outdoor as a child process, ensuring state between these is synchronised.

These responsibilities were separated into components that can communicate with the WebView2 layer and the web application. During startup select native APIs are exposed to the web application

via the WebView2 host object system, allowing it to interface with Windows to store user generated files and settings.

When started, the host first initialises and loads the web application; once complete, it starts Outdoor as a child process. The former ensures the user is given information about the state of the application while Outdoor is initialising, while the latter grants easy access to Outdoor's process handle.

As controlling Outdoor is done through its REST API, the web application contains most of the business logic, able function independently even without the host. After each component has successfully started, the web application establishes a connection with Outdoor, and the system is ready to use.

3 TECHNOLOGY

There are several different frameworks and technologies readily available for hybrid application development. At the start of this project multiple studies were conducted to evaluate candidates as well as the viability of the entire project on a high level.

This was especially challenging as hybrid applications rely on two vastly different class of technologies, each of which oversee responsibilities in unrelated environments: while web technologies only must interface with standard web components in browsers' heavily sandboxed environment, hybrid hosts must be able to interface with both the browser context and the operating system as well.

The findings of these internal studies are presented divided into two chapters specific to the type of technology inspected, with first their process and then their resulting technology choices described and introduced.

3.1 Host

There are several frameworks and technologies available for the role of the web host or wrapper. These aim to close the gap between native system applications, which have access to mature and powerful system APIs out of the box, and web applications, which are restricted to the browser's sandbox and generally have little to no access to system features and resources (though bridging APIs such as WebBluetooth, WebSerial, etc. have recently become more common, albeit still with scarce support).

One of the internal studies at the start of the project focused on creating a list of possible technologies and evaluating them against each other as well as against the core project requirements known at the time.

Several frameworks have been researched during this study, focusing on their strengths and weaknesses both on their own but also in the context of this project. These include technologies such as Electron, Tauri, WebView2, Flutter, and React Native.

After careful consideration of these, the decision to use the .NET based WebView2 was made, as it aligned the closest with the core technical requirements:

- As Outdoor is a Windows application, the UI application must support the same environment. In line with the design and architectural plans this would be the primary (if not the only) native environment supported, with the rest being served through browsers.
- Integration with the current development and build pipeline, which already use and depend on .NET libraries. This also helped avoiding inflating this project's and Outdoor's technology stack.
- During evaluation we found most hybrid frameworks (such as Tauri) to be using the C++ WebView library to achieve cross platform support, which already uses WebView2 when running on Windows [2].

Additionally other factors unrelated to the project's requirements were also part of the decision, some common between candidates:

One concern (mainly impacting Flutter and Tauri) was the introduction of languages and technologies that had no internal support within the company and the team. This lack of familiarity with these tools would have made development substantially more difficult and time consuming.

Another concern (impacting Electron and React Native) was their performance and tendency as NodeJS based frameworks to require several dozen third-party dependencies. While this alone is not necessarily a problem (the web part of the application will introduce these anyway), given the role and responsibilities of the host application it was more desirable to create application specific interfaces, rather than rely on general ones provided as JavaScript modules, and thus avoiding dependency hell. This decision proved to be helpful several times, as some resources easily accessed through .NET are only available via a combination of two or more third-party libraries.

3.1.1 .NET and Windows Forms

.NET is Microsoft's free, open source, and cross-platform development platform designed for building a vast array of different kinds of applications on top of a standardised runtime [3]. The main language used for the development of .NET applications (and the language used in this project as

well) is C# [3], a general-purpose object-oriented language. It provides several safety and convenience features such as type safety and garbage collection, and has provides great interoperability with its addon system, NuGet.

There are several application and UI frameworks under the .NET umbrella, although many are Windows specific [4]. One such UI framework is Windows Forms (WinForms for short), designed to build powerful desktop applications easily using a set of common UI controls, utilising the window (form) based approach. As in this case the native application is merely the “host”, the used UI framework does not matter, so Windows Forms was chosen for its simplicity and familiarity.

Another advantage of Windows Forms is its maturity. Although it has been superseded by other UI frameworks and more modern application types, it is still widely used and supported on several versions of Windows and by older versions of .NET.

For this reason, today’s .NET’s predecessor, the older Windows only .NET Framework 4.8 (C# 7.3) was used for the host application as Outdoor already directly depended on it, and it is guaranteed to be present by default on every machine running modern Windows as part of the operating system [5]. Out of the box support for common Windows versions was important, especially as industrial settings tend to lag behind with OS versions.

This choice of versions also ensured the project could integrate seamlessly into the already existing development and build pipeline in place, further reducing friction when developing a new component as part of a long-standing project.

3.1.2 Microsoft Edge WebView2

The Microsoft Edge WebView2 control is a relatively new technology released at the end of 2020, designed to allow developers to embed web content into native applications. It effectively replaces the old Internet Explorer based WebView control with the much more powerful Microsoft Edge runtime, providing feature parity with Chromium and additional APIs specifically designed for content embedding – such as with hybrid applications.

It can be utilised in a wide range of environments, such as .NET, .NET Framework, Win32/C++ and WinUI 2 and 3. Although at the time writing it is Windows only, Mac OS and Linux support is listed on the development roadmap [6].

In addition to providing ways to interact with web content (either local or remote), the WebView2 control also allows developers to expose native code directly to the web application with tight control over their scope. These APIs make it a perfect candidate for developing hybrid applications using a web UI backed by native code and position it as Microsoft's direct competitor against similar solutions in the hybrid development space, such as Electron.

Its execution model also differs greatly from others: the application utilising the WebView2 control acts as a host and uses the Control to interface with the underlying WebView2 system, which is running as a separate, independent process (Figure 5). Through this interface host applications can manage the instance's browser user data on their own, manage and alter web interactions, and share native classes with the JavaScript runtime allowing web code to directly call native methods.

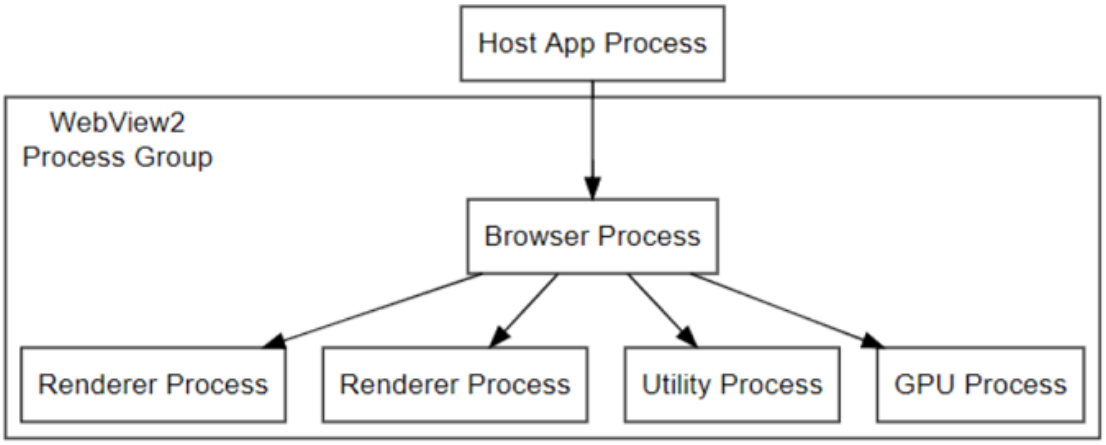

Figure 5. WebView2 runtime diagram (microsoft.com) [7]

An additional advantage is the Evergreen distribution model: on Windows 10 and above the WebView2 runtime is present by default and is maintained by the operating system, so applications utilising it do not have to distribute a specific version along with the application files (although there is an option to require and bundle a fixed version) [8].

This gets around the common problems of hybrid applications' distribution size being inflated by the browser runtime, and needlessly having (often the same version of) the same runtime installed

multiple times for different applications. Additionally, since the runtime is managed by the operating system, updates and security patches are automatically applied ensuring safe operation.

In this project's case when evaluated against Electron, a dramatic decrease in binary size was found that held up throughout the entire project: only 11Mb for a full build at the end of the project (including the entire web application), versus Electron's 142Mb when the application was still in its infancy. For context, just the chromium third-party licenses file bundled with every Electron application (9.79Mb in the test build at the time of writing) is almost as large as the entire distributable.

An additional aspect for its choice was the realisation that Outdoor already used the WebView2 system to embed web content such as maps in its currently existing interface, so its selection did not result in any new dependencies.

While a core component of modern Windows, the interface is not available in Visual Studio by default: these components are part of the `Microsoft.Web.WebView2` NuGet package and must be installed manually per project before use.

3.2 Web

There are a vast number of Web UI libraries, and year by year more emerge. However, there are only two that has been persistent and widely popular industry wide [9], and such can be considered the most viable for a hybrid application (consider documentation, support, ease of hiring): React and Angular. The second main internal study focused on selecting the most fitting one for the project.

When choosing frameworks, they should be carefully evaluated and chosen on a project-by-project basis. While some other technologies are obvious (JavaScript being *the* web programming language), web UI frameworks in particular can be highly specific to the given project and may not satisfy all requirements. This means the most important technical and design details should be compiled before this stage.

As an example, while developing the web application during this project, one major requirement coming up fairly early on was the ability to display graphs in “windows” that can be resized and reordered by the user, while keeping their state and active status (Figure 3).

This is not a concept native to the web world, so the implementation heavily depended on the chosen framework; React was found unable to handle scenarios like this, while Angular’s much more dynamic architecture was just perfect for the task.

Situations like these can also arise deep into a project’s lifecycle of course – by when switching to a different framework is not at all feasible – which further illustrates the importance of up-front thorough requirement collecting and planning.

Another aspect that should be considered – especially in large corporations – is internal support. In this project’s case we found most company web products to be using Angular, and as such have access to several internal toolkits, as well as a dedicated support team.

3.2.1 JavaScript and TypeScript

JavaScript is a lightweight interpreted scripting language, serving as the de facto programming language used for web development. All web browsers can execute JS code, and websites make heavy use of it to provide users with highly interactive interfaces. It is a prototype-based highly dynamic object-oriented language, offering great flexibility even during runtime.

TypeScript is a compiled superset of JavaScript maintained by Microsoft and introduces several new development time features to the language. It provides development time static typing, type notation and several other convenience features designed to make web development easier and safer [10].

By now TypeScript has largely “replaced” pure JavaScript for new and large projects, as projects grew larger and more complex, and the base language’s shortcomings become more apparent.

It is important to note however that these additions are strictly development time only, and type information and other metadata is stripped during compilation. As such great care should be taken

when working with data from external sources, as the incoming data may not match the expected type.

All web side code presented in this thesis has been written with TypeScript notation, as it ensures C# types can be directly and easily represented in examples, where switching between environments is discussed.

3.2.2 Angular

Angular is an open-source, TypeScript based web application framework developed by Google. It provides powerful built-in solutions for common tasks that allow it to stay largely self-contained and ensure each part of the application can take full advantage of the platform.

Angular's architecture differs from other frameworks: it is class based, rather than following the functional paradigm. Components of the application can be of different types: UI Components (simply components from now), Services, Modules, and more [11].

Components are reusable elements that usually make up a piece of the user interface. These are made up of three parts: the class controlling its behaviour and serving as its core, the HTML template which describes the component's structure, and the stylesheet which provides the CSS styles [12]. Although during development these are contained in different files, Angular seamlessly provides interactions between them. Events and state changes can be consumed via two-way data binding in the templates, which allow component methods and properties to be directly referenced from HTML.

The other integral part of Angular are Services. These provides overarching functionality to multiple components within the application and serve as the main mechanism for application wide state management. Each service is only (by default) instantiated once, so the same instance is available to every component, following the singleton pattern [13].

Consuming services is also extremely easy via dependency injection [14]. Components can include the services they require in their constructor, which is automatically resolved by Angular during compilation from just the TypeScript type information:

```
constructor(
    private _nemoApi: NemoOutdoorApiService,
    private _measurementService: NemoOutdoorMeasurementService,
    private _dialogService: AlloyDialogService,
    private _userPrefs: UserPreferenceService)
{
    //...
}
```

Figure 6. Example of Angular's dependency injection.

On a larger scale, components and services can be bundled into Modules. These can be used to isolate features and APIs into self-contained sections that can be shared across multiple applications.

Another advantage of Angular stems from its general architecture of internal component handling: unlike with many other frameworks where external, third-party modules and libraries are required for even common tasks; Angular includes solutions to common use cases with first-party libraries [15]. Using these even extremely complex scenarios encountered during this project were easily solved using only built-in tools or quick custom solutions.

The main reason for its selection however was company support. Keysight maintains an internal Angular component library, Alloy, that provides company specific controls and tools, in additional to ensuring all (including standard) components adhere to the company wide UI design standards. During evaluation this was the safest choice in this regard, as the library is developed by a dedicated internal team and is used by several services and products.

While an on-site team maintained a React version of this, ultimately Angular with its class- and services-based architecture was deemed more fitting for a hybrid application.

4 DEVELOPMENT

This section of the thesis describes the main milestones of hybrid application development with the previously described technologies, focusing on the most important technical challenges and features specific to them as encountered during this project.

Like the development of this project, this chapter is broken down into two main sections with the first one focusing on building the host application and interfacing with the WebView2 Control; the other focusing on the Angular web application and the challenges of building one application for multiple environments.

As the development of the native and web sides differ greatly not only in terms of technologies but also requirements, they both have vastly different toolsets – sometimes with their own (version dependent) quirks; for the sake completeness and reproducibility it is important to note the tools used for this project. The host application was developed in Visual Studio 2022 version 17.6.3 using C# 7.3 (.NET Framework 4.8), while the web application was developed in Visual Studio Code as a TypeScript 4.9 and Angular 15 project.

4.1 Host application

The purpose of the host is exposing a way for the web application to interact with native APIs and resources in a controlled manner and start and manage Outdoor as its child process if available. In this case, the former includes file operations for saving user generated files such as workspaces, used to save data windows layouts and configurations on the Monitoring page of the web application.

As discussed before in chapter 2.2, the host sits between the backend and the fronted. Its own internal design and architecture reflects this: the core of the application is built around a WebView2 control and supporting layer, with the rest of the functionalities separated into independent class modules designed to interface with it. This modular architecture works perfectly with the way native objects are exposed to the web context, while keeping the application itself very simple.

4.1.1 Initial set-up

As the entirety of the UI is web based, the host application is left with mostly management responsibilities. Due to this, only a single Form is used, with a WebView2 Control set to always fill its content area (Figure 7); no other UI controls are used or required.

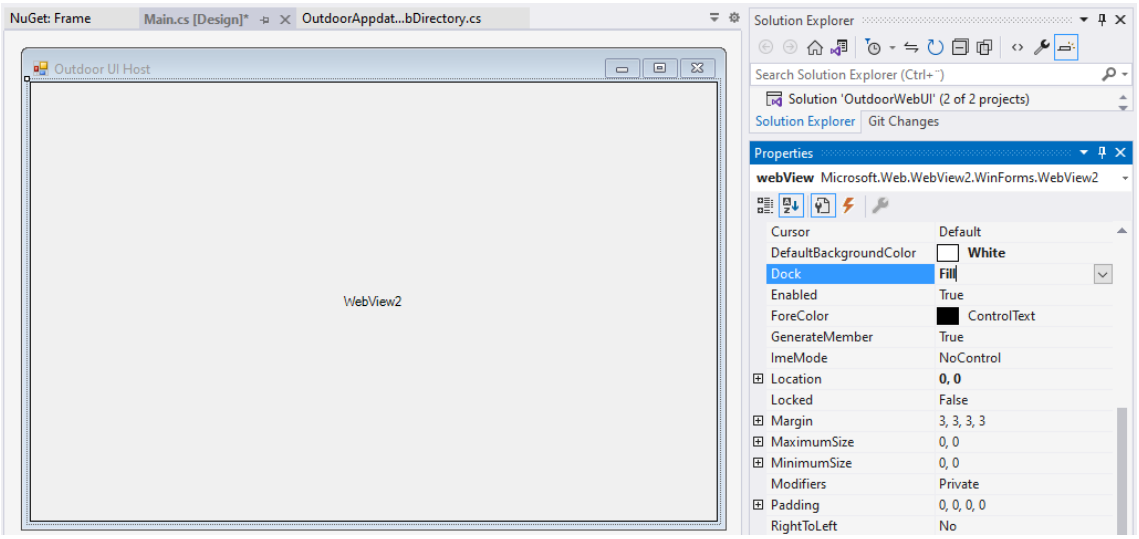

Figure 7. The host application's main form with the single WebView2 control set to Fill.

Outside of the Designer, setting up and initialising the WebView2 control differs substantially from how most other WinForms Controls are set up, as it is initialising an entire web browser in the background. Rather than being ready to use on application startup, it must be manually initialised first.

First, a new `CoreWebView2Environment` is created. This defines the version of the WebView2 runtime used, the location of the user data directory, and several other control options that are normally managed by browsers (such as display language or tracking prevention). In this case the Evergreen distribution model was used, meaning no specific WebView2 runtime is bundled with the application, so this option is ignored and set to `null`. However, the user data directory's location must be changed, as by default, it is the same as the compiled executable's location. After installation in production, this is likely a write restricted location (such as C:/Program Files) and will cause errors. An ideal place for this and other configuration files (and in this case other user generated files as well), is the application's automatically generated AppData directory, queried via `Application.LocalUserAppDataPath`.

Using this environment object, the control can be initialised by calling its `EnsureCoreWebView2Async` method.

After the initialisation has completed, the WebView2 instance can be further configured for hybrid application usage. By default, it is running assuming the role of a regular browser, with several functions that are undesirable in this use case and can be set on the WebView2 controls' `CoreWebView2.Settings` property.

Some of these options are cosmetic and can be disabled to accomplish the expected user experience of a native application (disable link previews, zoom, etc; common in web design but unexpected in native), while others are security related and control access to features that the application may or may not require (IPC messages, host objects), or would be otherwise dangerous to expose to users (such as the Edge DevTools).

Cosmetic options depend on the application and its use case; however, security related options should be always carefully considered. The `AreDevToolsEnabled` option should always be disabled in release mode as it exposes the standard browser developer tools including the debug console, allowing users to potentially modify the application or access its internals during operation.

Depending on the utilised resources, the `AreHostObjectsAllowed and IsWebMessageEnabled` options should be disabled as well when not used. These options control access to the main ways web code can communicate with the native application. Host objects are used to expose native APIs to the web context (see chapter 4.1.4), while web messages provide a simple messaging system between the two contexts. Both can potentially be exploited and should only be enabled when a trusted web application is running.

To define and control these options consistently from a single place, a dedicated static helper class `WebViewConfiguration` was created that can apply a preset configuration to any passed WebView2 instance. This is used at startup as the last step of the application's initialisation flow.

4.1.2 Web loader

There are multiple ways of loading the web application within a WebView2 host, depending on use-case and requirements. The easiest and the best suited method for hybrid applications is via the `SetVirtualHostNameToFolderMapping` method, which acts as a simple webserver that makes the contents of the specified directory accessible under the given virtual host name.

When specifying a host name for the method, the best practice is to use either the `.local` or `.localhost` top level domain [16], both to avoid any potential conflicts with real websites and for security, as these will never resolve outside of the local network or the host machine, respectively.

After setting up the directory mapping, the web application can be loaded by navigating to its index file by setting the `Source` property:

```
private void SetWebViewLocalSource(string UIPath = null)
{
    string path = UIPath ?? Constants.UIFilesPath;
    // Check if UI files exist
    if (Directory.Exists(path) && File.Exists(Path.Combine(path, "index.html")))
    {
        webView.CoreWebView2.SetVirtualHostNameToFolderMapping(
            Constants.VirtualHostName,
            path,
            CoreWebView2HostResourceAccessKind.Allow);
        //                        nemodesktop.localhost
        _url = new Uri($"https://{Constants.VirtualHostName}/index.html");
        webView.Source = _url;
    }
    else
    {
        MessageBox.Show(
            $"Core application files are missing. Reinstall the application to
repair the installation.",
            "Missing application files",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
        Close();
    }
}
```

For increased security of the host application, it may be desirable to intercept and prevent navigations and requests to external or non-approved sites. This is recommended, as the web application files are generally accessible on the user's computer and could be tampered with much easier than the compiled binary of the host application.

4.1.3 Starting headless

After loading the web application, the next step is starting Outdoor. This project is technically working with three applications (Figure 4): Nemo Outdoor, the hybrid host, and the web application. It is the host application's responsibility to synchronize state between these processes.

Before the start of this project, substantial work was done to completely de-couple the current UI from the business logic allowing Outdoor to be started as a headless service. When started in this manner, Outdoor runs with a REST API layer active on top of the standard core that allows it to be controlled after authentication.

Since Outdoor runs as its own independent application, the host must be able to start and manage its process. To achieve this, a two-part system was created: a bootstrapper that handles starting and capturing the process, and a watchdog that monitors it and ensures the host is always aware of Outdoor's state.

After the web application has initialised, the bootstrapper spawns a new process handle starting Outdoor (or recaptures an already existing process, if any); then transfers the process handle to the watchdog.

This structure introduces a problem however: when the main application and the UI host are running in different processes, one can fail without the other knowing, resulting in either an unresponsive UI or an idling backend. Handling these cases is the responsibility of the watchdog running in a separate thread within the host.

The host failing is unlikely as the WebView2 service also runs in its own separate processes that can fail independently without crashing the Form. This event can be captured via either the `CoreWebView2.ProcessFailed` event raised when any of the application's WebView2 processes fail, or with the `CoreWebView2Environment.BrowserProcessExited` event raised when the main browser process exits[1].

[1] There are a lot of nuances to when and how these events are raised that is beyond the scope of this thesis. See the documentation [9] for more.

In the case of Outdoor failing, the event can be caught by simply subscribing to the `Process.Exited` event (`EnableRaisingEvents` must be set to `true` on the process handle).

In these scenarios the watchdog can passively manage these processes, however it is still cannot synchronise their states. The backend application will likely still be initialising by the time the UI is ready, and the host has no inherent way of knowing when it is ready; in Outdoor's case a cold start can be considerably more than just a few seconds.

This can be solved by utilising .NET named pipes. These provide a communication channel between processes and allow late connections; perfect for this use-case. In this case they are configured to allow bi-directional communication so status updates and commands may be transmitted from either side (Figure 8).

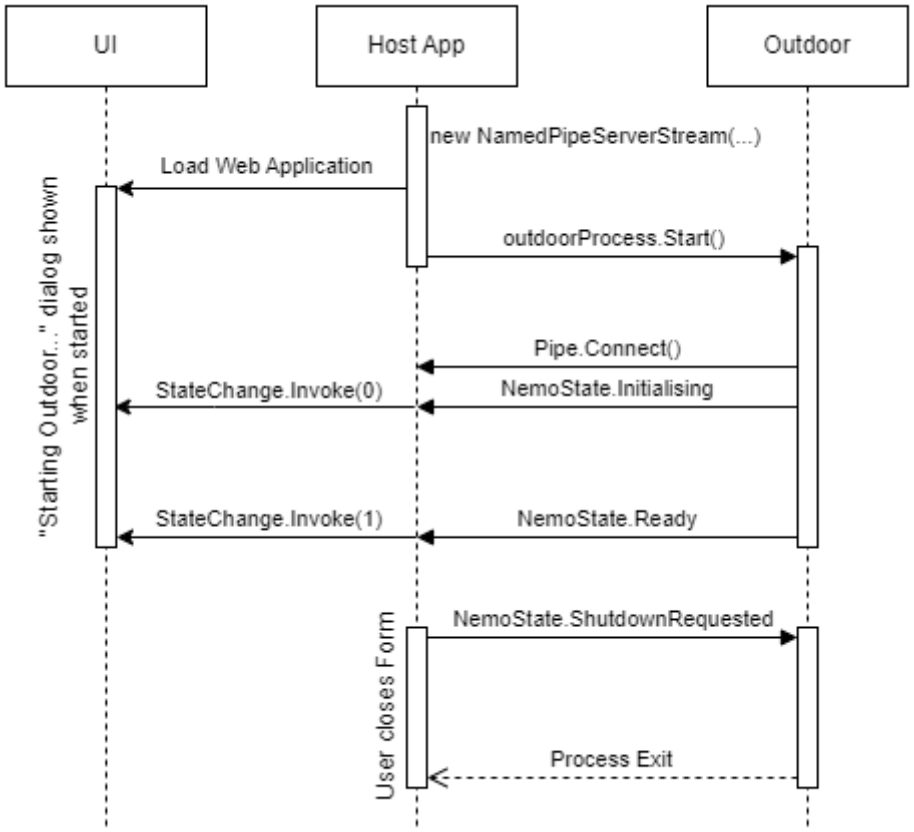

Figure 8. Sequence diagram of state synchronisation between the three processes.

For example, the web application will display a loading dialog while waiting for Outdoor to complete initialising. When Outdoor connects to the pipe and transmits updates, the host can relay them to the UI via the WebView2 messaging API to change status messages providing the user with

information while waiting. Once the `Ready` status has been received, the dialog is closed, and the application is ready to be used.

Similarly, when the user closes the UI host, the form's `FormClosing` event sends the `ShutdownRequested` command to Outdoor so it can save configuration and shut down safely, rather than being killed abruptly by the host.

The transmitted information is kept simple, so the messages exchanged can be represented as signed `int32` values that are mapped to a common state enum (Figure 9). Values above 0 are reserved for state updates, while values below zero are instructions.

```
private enum OutdoorState
{
    // Status values from the Outdoor main process
    Initialising = 0,
    Ready = 1,
    // Commands and status values sent to the Outdoor main process
    ShutdownRequested = -1,
}
```

Figure 9. `OutdoorState` enum used to map command and statuses to int values.

4.1.4 Exposing native APIs

Exposing native APIs to the hosted web application is fairly straightforward in C#, however it still comes with caveats. As the WebView2 instance is running in a separate process not owned by the host application (the WebView2 Control on the form is merely an interface for interacting with it), it can only access objects from the host if they are exposed to the COM.

Objects can be exposed to the WebView2 runtime by applying the `ComVisible` and `ClassInterface AutoDual`[2] attributes (Figure 10) and marking them `public`. By adding these attributes, the objects' COM interface will be automatically generated at compile time so the exposed objects can be seen by the control (normally the burden of defining and registering these interfaces would fall on the developer, but the .NET C# compiler automates this).

[2] At the time of writing AutoDual is marked as Obsolete, however it is still used in the official documentation.

```
[ClassInterface(ClassInterfaceType.AutoDual)]
[ComVisible(true)]
public class FormFullscreenToggle
{
    //...
}
```

Figure 10. Class with COM attributes applied.

Exposed classes should be largely self-contained and able to function independently from the rest of the host application, only exposing choice members that are designed for interoperability. While making a class available to COM automatically exposes all public fields and members, their visibility can be set by applying the `ComVisible` attribute individually.

Design of these classes should be more deliberate and restrictive than usual; whereas with compiled binaries the risk of misuse is relatively low (whether maliciously or unintentionally), interfaces exposed to the browser runtime are inherently unsecure.

Arguments of browser facing methods should be validated and sanitised, and their responsibilities restricted [17]: for example, the class used for managing user generated files is deliberately designed to manage the contents of a specific directory only, instead of exposing the entire Windows File API to the browser context. In contrast, the generic Windows File API could be used by the entire application unrestricted, and since JavaScript is not compiled any arbitrary code could use these APIs without restriction.

Once a public facing interface has been constructed, it can be exposed to the WebView2 control via the `AddHostObjectToScript` method (Figure 11) at any time. The first parameter required is the host object's name: this is the key that the object will be accessible by in the browser context (see chapter 4.2.1).

```
var hostForm = webView.FindForm();
var toggle = new FormFullscreenToggle(hostForm);
webView.CoreWebView2.AddHostObjectToScript("fullscreenToggle", toggle);
```

Figure 11. Example of exposing a class as a host object.

Once no longer useful, host objects should be removed for security reasons via the `RemoveHostObjectFromScript` method.

Despite its ease of use, the host object system still has some quirks to be aware of, some of which will surely be solved and streamlined in the future – as this technology is still relatively new at the time of writing – but are important to note.

Debugging is supported as expected, but in some cases identifying the cause of an exception can be difficult because of the inter-process architecture. For example, missing COM attributes and incorrect access level declarations are not detected at compile time but will throw exceptions during runtime: `ArgumentException` ('Value does not fall within the expected range.') can be particularly difficult to make sense of, but almost aways corresponds to an access issue.

Another quirk concerns how objects are handled on the JavaScript side (see chapter 4.2.1 for details). Exposed methods and properties should ideally return simple C# types, as these mostly have direct representations in JavaScript; for example the C# `string[]` is automatically cast as a `Array<string>`, and vice versa. More complex types such as Lists are represented as proxy objects, and as such are much more difficult to work with than with a native type (there is also some performance cost to doing frequent marshalling). Considering this, if interaction with a complex type is required it may be easier and more efficient to create and expose methods for manipulation; otherwise simply cast results into a simple type (such as via `List<T>.ToArray()`) before returning.

Most other language mechanism work out of the box however: for example, the C# `EventHandler` is cast as a standard JavaScript `EventTarget`, and asynchronous methods are await-able like native JavaScript async functions.

4.2 Web application

The development of the web part of a hybrid application largely followed the same course as a regular web application, with the main difference being the introduction of native system resources.

The focus here was on the two main technical requirements: ensuring measurement data streamed by Outdoor is processed and distributed to consumer UI elements (performance) and ensuring the application can function on a high level in an environment agnostic manner (cross-platform design) while still utilising native APIs whenever available.

The web application is essentially built on top of two core Angular services overseeing the connection between it and Outdoor (Figure 4). The `NemoOutdoorAPIService` abstracts away all network communications, exposing simple methods corresponding to different REST API endpoints provided by Outdoor, to the rest of the application. It also provides connection and state information and performs authentication and automatic response validation. Building on top of this service, the `NemoMeasurementService` manages functionality related to live measurements and data queries.

Building on these core services (along with several others), the rest of the application was constructed according to current Angular application development best practices.

The only otherwise notable difference is the utilisation of Keysight's internal Angular component library, Alloy, that provides several company specific controls and ensures it adheres to the company wide UI design standard.

4.2.1 Consuming native APIs

The WebView2 control exposes its own APIs within the browser context, accessible on the global `window` object[3] at the `window.chrome.webview` key. This provides access to the WebView2 API in general, as well as the host objects exposed by the host application.

[3] At the time of writing no TypeScript typing package exists, so type extensions need to be manually created following the official documentation at https://learn.microsoft.com/en-us/microsoft-edge/webview2/reference/javascript/webview

The `webview` object extends the `EventTarget` class to provide listener subscriptions used for inter-process messaging. Message listeners can be assigned as standard and incoming messages can be intercepted with the `message` listener type.
Sending messages to the native side is done via the `postMessage` method, which automatically serialises the passed message object into a JSON string.
Objects exposed by the host application (as described in chapter 4.1.4) can be accessed by the key name given during registration as `window.chrome.webview.hostObjects["keyName"]`, although differently than how other technologies (like Electron) would handle them.
Rather than appearing as modules, host objects are accessed through asynchronous Proxy objects. These proxies are directly backed by the C# native code while also exposing several utility methods to facilitate interactions (for example providing a mechanism to choose if a remote or local method or property should be called when they share the same name), otherwise they behave identically to regular JavaScript Proxies.

Host object methods and properties are added onto the proxy for direct access, although by default every invocation is done asynchronously with Promises. This is mostly due to marshalling being relatively expensive, and given JavaScript's single-threaded nature it could block the main thread, impact performance and responsiveness; nevertheless is it possible to perform synchronous calls as well via the `sync()` method (or by accessing the host object via `hostObjects.sync["keyName"]`).

```
window.chrome.webview.hostObjects["fullscreenToggle"]["SetFullscreenMode"](state)
.then(()=>{
    // Method returned on host side
});
```

Figure 12. Host object method invocation.

Several helper methods interacting with this API have been written and packed as an Angular module to aid development. Most interfacing with the WebView2 controls is done through this addon module, ensuring that the application behaves consistently regardless of environment.

One of the more frequently used methods created to provide a safe context for- and facilitate error handling originating from host objects is the `WV_SafeExecute` (Figure 13). While in essence it is merely a slightly smarter try-catch block, this pattern has proved to be invaluable during development and testing, as exceptions thrown by host object errors often originate from deep

within the WebView2 API implementation which can be difficult to make sense of (the message can be confusing and the stack-trace misleading or outright not useful).

```
/**
 * Provides a safe context for WebView hostObject interactions. Exceptions
will be re-thrown with a valid application context stacktrace.
 * @returns
 */
export function WV_SafeExecute<TReturn>(func: () => TReturn) {
    if (isRunningInWebView2()) {
        try {
            return func();
        }
        catch (e) {
            throw WEBVIEW_INTERACTION_ERROR(e);
        }
    }
    throw WEBVIEW_UNAVAILABLE_ERROR();
}
```

Figure 13. SafeExecute method implementation.

As this method wraps any unhandled exception occurring in the passed method into the custom error `WEBVIEW_INTERACTION_ERROR` to provide accurate stack-trace and an immediately obvious error message, it is meant to be used to wrap host object interactions directly to avoid intercepting non WebView2 related exceptions.

Additionally, this method ensures no undefined behaviour occurs from accidental calls to native APIs when the application is not running in a WebView2 instance.

4.2.2 Cross-environment design

The web application was designed to run in either a regular web browser or as native application through the WebView2 host from the start. While most of these two environments are identical, the systems available differ, which requires the application to be aware of the runtime environment and its characteristics and utilise these sometimes vastly different systems appropriately.

Due to this, it is imperative that proper feature detection is done before any environment specific calls are attempted. Although normally platform detection is heavily discouraged and tends to be

extremely unreliable, in WebView2's case it can be reliably performed by checking if the `window.chrome.webview` object exists.

A strong example for this cross-platform abstraction is the way the application handles user generated files and settings. This data must be saved to disk, for which the mechanism changes depending on the environment. In this case, only one target platform – Windows – supports a real filesystem, while browsers only have the `localStorage` API available which only allows string key-value pairs to be saved.

Identifying systems that need to function in different environments early is paramount and should be treated separately from the rest of the application. During this project these systems were designed to be as modular as possible, generally providing a single interface the rest of the application depends on, while having smaller modules automatically performing the work in the background depending on environment.

Angular's class-based architecture is perfect for these kind of situations as it heavily encourages building modular systems and classes by default.

User generated data files are categorised by type, so that a resource directory only contains a specific file type. Therefore, resource directories can be defined by a common generic interface that returns the specified type of file (Figure 14). For example, when reading a workspace file from a resource directory bound to the workspaces folder, the returned file is automatically cast to the expected data type so the monitoring page can correctly load it.

```
export interface IResourceDirectory<TFile extends VirtualFile> {
    readonly onDirectoryUpdate: Subject<string>;
    get directory(): FileTypeDirectoryMap;
    get source(): "browser" | "system";
    get size(): number;
    get maxSize(): number;
    getFiles(): Promise<string[]>;
    readFile(name: string): Promise<TFile>;
    save(name: string, contents: TFile['data']): Promise<void>;
    delete(name: string): Promise<void>;
}
```

Figure 14. Environment agnostic interface describing how user generated files can be accessed.

For easy access the resource names are synchronised across `localStorage` key names and native system directories, so entries in the `FileTypeDirectoryMap enum` correspond directly to their location regardless of environment (for example `FileTypeDirectoryMap.Workspaces` evaluates to “workspaceFiles”, which is either a key in `localStorage` or a directory in the application’s app data folder).

Using this, two classes were created for handling resources in the two separate environments: `LocalStorageResourceDirectory` for browsers and `WebViewResourceDirectory` for the native host.

When Angular instantiates the file system service, the constructor checks for the WebView2 API and decides which source to bind to. As the fields in the service are defined to accept the interface instances, depending on the environment either a `WebViewResourceDirectory` or `LocalStorageResourceDirectory` instance will be slotted into the fields. This way, for the rest of the application the underlying mechanism does not matter.

A helper method was created to facilitate automatic environment dependent initialisation and feature selection (Figure 15). In this project the hybrid application environment was treated as the “main” environment with the stock browser version considered a secondary – unit specific – control page, hence the fallback designation.

```
/**
 * Executes different methods depending on the current runtime (WebView2 or
regular browser).
 * Use to provide fallbacks to when native commands through WebView2 are not
available (e.g.: saving a file in Window -> save to browser localStorage).
 *
 * @param webView Method to run if the application is running in a WebView2
frame
 * @param fallback Method to run if the application is running in a regular
browser
 */
export function ExecuteEnvironmentSpecificImplementation(options: { webView:
() => void, fallback?: () => void })
{
    if (isRunningInWebView2() == true) {
        WV_SafeExecute(()=>{
            options.webView();
        });
    }
    else if (options.fallback) {
        options.fallback();
    }
    else if (options.fallback == undefined) {
        console.warn("WebView unavilable; fallback method not specified.");
    }
}
```

Figure 15. Helper method for automatic environment selection.

A deep understanding of how these different underlying systems work is important, as their characteristics heavily influence the design of shared APIs. In this case there are multiple quirks that the interface must support stemming from the different storage systems, some small and some large enough to be its own system.

A simple example is the method return types on the `IResourceDirectory` interface shown on Figure 14; all methods return their results in promises. While the local storage browser API is synchronous, the WebView host objects primarily work asynchronously (although loading files is relatively rare, blocking the main thread for even a second is not desirable). Therefore, it is easier to compromise and use promises in both cases.

The much more complex case pertains to storage size limitations. Modern browsers allocate a maximum ~5Mb of storage per origin [18] for local storage, which is a hard limitation and low enough that the user must be aware of (although these files take up little space, users are expected

to utilise them heavily, so hitting this limit is conceivable). For this reason, when the application is running in a web browser, an extra piece of UI is shown in a utility panel informing the user of the total, currently used, and remaining storage space available, with special dialogs for when no more is available.

This limitation however does not exist in Windows where these files are saved into the user's AppData folder, and the available storage space is *technically* the disk's remaining capacity *(don't)*.

While the size constraint is important in a browser, it is negligible when using the native host. As a result, the `LocalStorageResourceDirectory` instance correctly calculates the `size` and `maxSize` values, while `WebViewResourceDirectory` simply safely return -1 and infinity, respectively. This ensures that existing checks validate correctly while still appropriate to the environment: when checking if a file can fit in the available space (file size + current size < maximum size) the result will be correct regardless of source.

Normally a size calculation like this would be dangerous, but the files saved are a few dozen kilobytes at most, so the system wide storage requirement is negligible (during testing the total space used was estimated to be around 30 megabytes for several hundreds of complex files).
The primary reason for this hack stems from user experience and the way Windows applications behave: users don't expect to see storage indicators on native applications, as it evokes the feeling of using a cloud based service where storage quotas are limited (for example, Microsoft OneDrive shows cloud storage information in their Windows application, but not the local disk usage).

4.2.3 Performance and multithreading

A web application's constrained access to system resources does not only apply to native operating system APIs but affects general performance as well. While browser engines are heavily optimised, web contexts are limited by design, with one of their main bottlenecks being their single threaded design. This is not limited to just general JavaScript code; every aspect of a web page (application) is executed on the same thread within the renderer process, including the HTML and CSS parsers.

Due to this design, overly processing heavy tasks can have detrimental effects on the entire application and affect the UI's responsiveness. While this is not generally an issue, as web

applications become more complex this limitation becomes more apparent, especially in processing heavy applications. This problem can be further exacerbated by frequent updating of UI elements (in this context meaning larger edits to the DOM), something an application such as this is bound to do.

This contrasts with operating system native frameworks – such as Windows Forms – where UI components are running on their own shared thread, and resource heavy application logic can easily be delegated to background threads to avoid blocking the UI thread. In Windows Forms case these threads can not only still communicate with each other, but also invoke UI components on the main thread and perform cross-context calls.

Modern JavaScript provides an interface similar to these worker threads in the form of the Web Workers, although with significant differences. Workers are entirely sandboxed into their own execution contexts, separate from the rest of the web application. They cannot access resources on the main thread or invoke UI components; the only way of communication is via sending messages through a shared IPC channel.

Most of the application performs simple UI updates or minimal processing, such that this bottleneck is never encountered. The exception to this is the Measurement Monitoring page, where potentially more than a dozen graphs may be updated at the same time, rearranged or their data sources updated by the user; incoming live measurement data must also be processed and dispatched to the correct data window. All these concurrent responsibilities make this part of the application the most resource intensive and bring the importance of performance optimisations into the spotlight.

To handle all these tasks, a new Angular service – `NemoMeasurementService` – was created. Initially all these tasks were handled directly in this service, but it quickly became apparent that the processing required was negatively impacting application performance and caused stability issues.

The tasks in this action flow can be divided into two distinct categories: graphing and processing. Graphing is handled by the HighCharts library and as such bound to the main thread. Processing on the other hand includes maintaining a WebSocket connection per graph, parsing and conversion of the incoming data, and handling of Outdoor's custom measurement subscription format; none of these utilises APIs only available on the main thread, or require access to other parts of the application.

Therefore, most of the processing can be safely decoupled from the measurement service and moved to a dedicated worker. The design of this system loosely follows the example of the WebView2 host objects, with a proxy and backing-object approach: each connection consists of a `MeasurementMonitorProxy` and a `MeasurementMonitor` instance (Figure 16).

The measurement service provides several Outdoor specific utility methods as well as manages its background worker thread and oversees the creation of monitors. When the user navigates to the measurement monitoring page, the service automatically starts the background worker; once it is ready data windows can request their own monitor instance and start listening to incoming data.

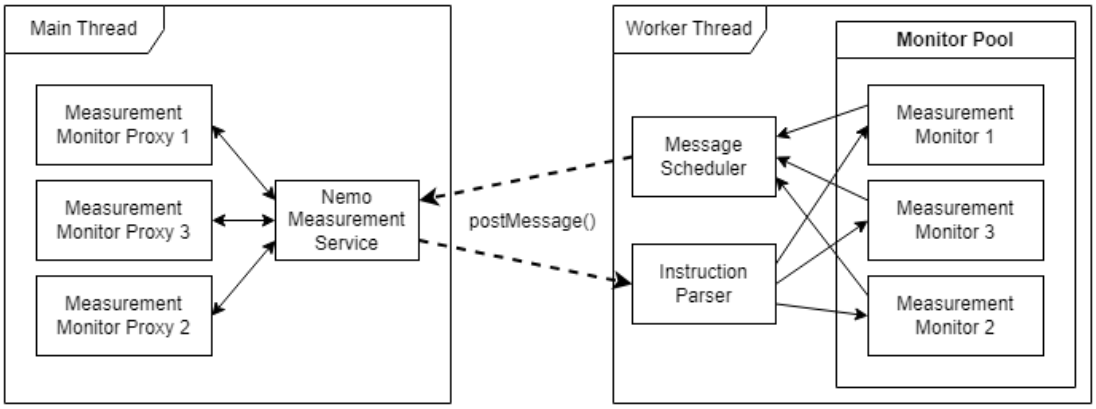

Figure 16. The Measurement Service's worker architecture.

`MeasurementMonitor`s exist only on the worker thread and create and manage their individual WebSocket connection with Outdoor, parse and transform incoming measurement data, and manage measurement subscriptions (devices, metrics). Proxies exist on the main thread and act as the public interface of the backing monitor; their methods create and send instructions that are transmitted to the underlying worker.

```
interface INemoMeasurementMonitorProxy {
    readonly id: string;
    readonly disposed: boolean;
    readonly data: Subject<NemoMonitorDataFrame>;
    readonly connected: BehaviorSubject<boolean>;
    readonly error: BehaviorSubject<NemoMonitorError | null>;
    reconnect: () => void;
    dispose: () => void;
    subscribe: (entries: NemoSubscriptionItem[]) => void;
    unsubscribe: (entries: NemoSubscriptionItem[]) => void;
}
```

Figure 17. Measurement monitor proxy's interface.

Proxies and their backing monitors communicate through the IPC channel exposed between the main and worker thread. To keep the two instances synchronised and connected, a random UUID is created upon instantiation which is then shared by both sides and is appended to every message sent. For example, when a subscription is added via a proxy's subscribe method, it is wrapped into an instruction (Figure 18) that is then sent to the worker via the `postMessage` method. On the worker's side, a central instruction parser looks up the corresponding monitor instance and passes on the message's contents to the appropriate monitor method.

```
{
    id: "8e898901-3b4c-467c-af09-03973261869a",
    subs: {
        add: [
            {
                device: 102,
                metrics: [
                    "Nemo.LTE.CELLMEAS.Cell.lte_received_quality.serving"
                ]
            }
        ]
    }
}
```

Figure 18. Example message of `NemoWorkerUpdateMonitorInstruction` type. The `id` value binds the message to a specific monitor instance.

While the worker maintains a list of its active monitor instances, the application level `NemoMeasurementService` does not. Each proxy independently subscribes to the worker instance's message event with its own event handler that catches incoming messages with its own ID, ignoring the rest.

Due to this two-part design, the C# dispose pattern was implemented to facilitate management of monitors across threads. When no longer needed (for example when the user closes a data window), calling the dispose method on a proxy will immediately disable it and send a dispose instruction to the backing monitor as well, which is then gracefully shut down and destroyed.

While this system provides an immediately noticeable performance improvement, there are still other problems that become prevalent when a large number of monitors are active: the IPC channel between the main thread and the worker can become clogged as the streamed measurement data throughput increases, causing data windows to constantly update their graphs at irregular intervals, which can significantly impact performance (while measurement data is streamed at regular intervals, it is only consistent within a single monitor; multiple monitors may receive their own data frames on a schedule so that one or more graphs are redrawing at any given time, which is a rather expensive process).

Although not fully implemented during this project (but represented on Figure 16), this problem can be solved by implementing a central message scheduler on the worker thread that all measurement monitor instances use to queue their messages, rather than individually sending them as soon as possible. This system combines messages from different monitors into one object that is then periodically (for example on a ~1 second loop) sent to the main thread where the proxies simply pick out their own events. This ensures that graphs update in batches and less frequently.

Additionally, other performance savings were introduced as well. For example, each workspace may have multiple tabs, with only one tab visible at a time. Data windows expose their own API that includes an event (among other things such as setting window titles, icons, controls) for detecting when the window's content is visible on screen using the intersection observer API. Using this, graphs that aren't in the current tab or visible automatically pause updates, freeing up both general performance and the IPC channel's bandwidth by instructing monitors to temporarily pause their streams as well.

5 CONCLUSION

Overall, the project has accomplished all goals and expectations. The user interface refresh was an internal success and established several future aspirations for the application's flow. On the technical side the three-part architecture proven to be viable, and the hybrid host lived up to all expectations. Although it had limited functional responsibilities and a considerable part of its capabilities were not connected through the REST API or were not production ready (due to this being an exploratory prototype, missing features were not implemented in Outdoor's REST API before verifying the development cost would be justified), it exceeded the expectations initially defined for the prototype.

Development wise, using WebView2 proved to be easier to use than anticipated and – at least in this project's case where the native UI is already separated and uses a service layer to access features – would not introduce any large architectural changes or technical challenges.

The web application in general was also a success, as it proved not only that it is a viable replacement for a native framework, but it also demonstrated that it could function in various environments while providing a responsive and straightforward user experience. It also further affirmed the value of Outdoor REST API.

With computers in general becoming more and more powerful and web applications become more mainstream and optimised, the performance difference between web and native starts to blur. Although for extremely data heavy applications it can be challenging to maintain responsiveness and general performance (considerable time was spent on the live measurement graphing pipeline), with careful designing and optimisation it is possible to eliminate most if not all these problems. By the end, it was functioning with similar performance to the native UI, supporting about a dozen live charts being displayed and updated with dozens of unique datapoints every second. With further optimisations – such as the previously mentioned but not implemented message scheduler – this would have likely reached near identical performance.

Still however, this remains an area web application continue to struggle with.

On the native host side, while in this project the host application was written in C# and is an independent application from the backend, considering the described development process and the fact that the WebView2 library has bindings for other frameworks and C++ as well, it is a perfect candidate even for applications where the business logic is more tightly coupled with the UI within the same process.

It is important to note however, that not every application benefits from this. Applications where a simple user interface with basic user controls is enough, it is more worthwhile to stick with native frameworks.

The real beneficiaries of a web-based UI are applications where more complex, multi-page interfaces with advanced controls and user flows are required; web application frameworks were designed specifically for these scenarios, and generally provide more ready solutions to common problems.

An additional use case for such hybrid applications is cross platform support, as sharing a common interface in the form of a web application can drastically reduce development time and cost where supporting multiple platforms is necessary. Although WebView2 is currently only available on Windows, once MacOS (and Linux) support reaches general availability, it will be perfectly suited for such use case.

In conclusion, the project achieved its development objectives and produced a powerful prototype that more than demonstrates the viability of upgrading legacy applications, to a modern web-based user interface using WebView2 as a host. Even if not directly built on this prototype, I expect a similar hybrid application eventually reaching release using the knowledge acquired during this project.

This project was also personally invaluable, as my first time working in a professional environment. Working in a team with several other developers of different disciplines to understand the design of such a large application as Outdoor to create a successful prototype have taught me several lessons that will no doubt stay with me throughout my career.

It was also fun and valuable to get to work on a project bringing two such wildly different aspects of development together; each of which I have worked with separately before.

Considering the learnings and experiences I have acquired during my time with this project, I firmly believe this type of hybrid application development will expand even more in the future and become a core discipline of the industry; and as such I am happy to now have considerable experience with it.

6 REFERENCES

[1] S. Singh, “Microsoft Teams: Advantages of the new architecture,” 27 03 2023. [Online]. Available: https://techcommunity.microsoft.com/t5/microsoft-teams-blog/microsoft-teams-advantages-of-the-new-architecture/ba-p/3775704. [Accessed 21 05 2024].

[2] webview project contributors, “Webview Readme,” 29 11 2023. [Online]. Available: https://github.com/webview/webview?tab=readme-ov-file#windows. [Accessed 23 05 2024].

[3] Microsoft, “Introduction to .NET,” 10 01 2024. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/introduction. [Accessed 06 06 2024].

[4] Microsoft, 16 05 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/apps/get-started. [Accessed 06 06 2024].

[5] Microsoft, “.NET Support Policy,” 29 05 2024. [Online]. Available: https://dotnet.microsoft.com/en-us/platform/support/policy#dotnet-framework. [Accessed 06 06 2024].

[6] Microsoft, “WebView2 Roadmap,” 21 06 2023. [Online]. Available: https://learn.microsoft.com/en-us/microsoft-edge/webview2/roadmap. [Accessed 22 05 2024].

[7] Microsoft, “Process model for WebView2 apps,” [Online]. Available: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/process-model?tabs=csharp#processes-in-the-webview2-runtime. [Accessed 14 04 2024].

[8] Microsoft, “Distribute your app and the WebView2 Runtime,” 12 05 2023. [Online]. Available: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/distribution. [Accessed 20 05 2024].

[9] Stackoverflow, “2023 Developer Survey - Web Frameworks and Technologies,” 05 2023. [Online]. Available: https://survey.stackoverflow.co/2023/#web-frameworks-and-technologies. [Accessed 21 05 2024].

[10] Microsoft , “TypeScript for JavaScript Programmers,” [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. [Accessed 06 06 2024].

[11] Angular, “Introduction to Angular concepts,” [Online]. Available: https://v17.angular.io/guide/architecture. [Accessed 06 06 2024].

[12] Angular, “Components,” [Online]. Available: https://angular.dev/guide/components. [Accessed 06 06 2024].

[13] Angular, “Creating injectable services,” [Online]. Available: https://angular.dev/guide/di/creating-injectable-service. [Accessed 06 06 2024].

[14] Angular, “Understanding dependency injection,” [Online]. Available: https://angular.dev/guide/di/dependency-injection. [Accessed 06 06 2024].

[15] Angular, “What is Angular?,” 15 08 2023. [Online]. Available: https://v17.angular.io/guide/what-is-angular#what-is-angular. [Accessed 06 06 2024].

[16] S. Cheshire and M. Krochmal, “Multicast DNS - RFC6762,” [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6762#appendix-G. [Accessed 22 04 2024].

[17] Microsoft, “Develop secure WebView2 apps,” 02 07 2023. [Online]. Available: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/security. [Accessed 12 05 2024].

[18] MDN contributors, “Storage quotas and eviction criteria - Web Storage,” 25 10 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria#web_storage. [Accessed 13 05 2024].

[19] Microsoft, “Handling process-related events in WebView2,” 17 07 2023. [Online]. Available: https://learn.microsoft.com/en-us/microsoft-edge/webview2/concepts/process-related-events?tabs=dotnetcsharp. [Accessed 20 05 2024].

[20] Microsoft, “Introduction to Microsoft Edge WebView2,” 24 05 2024. [Online]. Available: https://learn.microsoft.com/en-us/microsoft-edge/webview2/#hybrid-app-approach. [Accessed 27 05 2024].