Tampere University

Tuomas Pekkanen

PACKAGING COMPLEX WEB CLIENT IN EASILY EMBEDDABLE SOLUTION

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Tuukka Toivonen
Terhi Kilamo
May 2022

ABSTRACT

Tuomas Pekkanen: Packaging Complex Web Client in Easily Embeddable Solution
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
May 2022

Complex web applications require significant effort to be embedded by third parties to external websites. M-Files Corporation wants to make embedding the M-Files web client effortless for third parties. The existing supported embedding methods include using platform-specific add-ins that cannot be used outside the platform they are custom-made for. Third parties embedding M-Files to their own website would need technical skills and significant implementation effort to match the behavior of the add-in solutions. This thesis creates a generic add-in that can be effortlessly embedded into practically any external site.

Creating a generic solution requires that the solution supports the majority of modern browsers including their older versions. Browsers support varying versions of ECMAScript and not all browsers support all the features. Transpiling and polyfills are presented as a tool for ensuring compatibility with browsers by transforming the code to an older version of ECMAScript and including the browser's missing features in the solution code.

To implement the solution a framework is chosen. Frameworks are compared on the effort required to embed the result they create, their longevity, browser compatibility and flexibility with other tools. The framework must support transpiling and polyfills. From these criteria, Lit is chosen as the framework.

A prototype project is made with Lit to embed M-Files web client. The solution combines script-based embedding with an iframe to enable easy configuration. Website administrators can configure the solution with HTML element attributes and end users with a graphical configuration panel. To make including the solution to a website effortless, the created solution is bundled into a single script file that needs to be added to the third party website. In total 2 lines of code and minimal technical skills are required to embed the created solution.

Keywords: embedding, application, iframe, ECMAScript, JavaScript

TIIVISTELMÄ

Tuomas Pekkanen: Monimutkaisen verkkosovelluksen pakkaaminen helposti upotettavaan ratkaisuun
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Toukokuu 2022

Monimutkaiset verkkosovellukset vaativat huomattavaa vaivannäköä kolmansilta osapuolilta, jotka haluavat upottaa ne omille verkkosivuilleen. M-Files Oy haluaa tehdä M-Files verkkosovelluksen upottamisesta mahdollisimman helppoa kolmansille osapuolille. Olemassa olevat ratkaisut perustuvat alustakohtaisiin lisäosiin, joita ei voida käyttää sen alustan ulkopuolella jolle ne ovat luotu. Kolmannet osapuolet jotka upottavat M-Filesin verkkosivuilleen tarvitsevat teknisiä taitoja ja merkittävän työpanoksen luodakseen vastaavan ratkaisun, kuin olemassa olevat lisäosat. Tässä työssä luodaan ratkaisuksi yleiskäyttöinen lisäosa, jonka voi vaivattomasti upottaa käytännössä mille tahansa ulkoiselle verkkosivulle.

Yleiskäyttöisen ratkaisun luominen vaatii sen toimimista valtaosalla nykyaikaisista selaimista mukaan lukien niiden vanhemmilla versioilla. Selaimet tukevat vaihtelevasti ECMAScriptin versioita eivätkä kaikki selaimet tue kaikkia ominaisuuksia. Transpilaus ja polyfillit esitetään työkaluina, joilla varmistetaan yhteensopivuus selainten kanssa muuttamalla ohjelma vanhempaan ECMAScriptin versioon ja sisällyttämällä selaimen puuttuvat ominaisuudet luodussa ratkaisussa.

Ratkaisun toteuttamista varten valitaan kehysympäristö. Työssä verrataan kehysympäristöjä niiden luoman ratkaisun upottamiseen tarvittavan vaivannäön, niiden pitkäikäisyyden, selaintuen ja muihin työkaluihin sopeutuvuuden perusteella. Kehysympäristön täytyy tukea transpilausta ja polyfillejä. Näiden kriteerien perusteella Lit valikoitui kehysympäristöksi.

Litillä luodaan prototyyppi, joka upottaa M-Files verkkosovelluksen. Tämä ratkaisu yhdistää ohjelmointiperusteisen upottamisen iframen kanssa saavuttaakseen helpon konfiguroinnin. Verkkosivujen ylläpitäjät voivat konfiguroida ratkaisun HTML elementtien attribuuteilla ja loppukäyttäjät vastaavasti graafisesta konfiguraatiopaneelista. Jotta ratkaisun lisääminen verkkosivulle olisi vaivatonta, se paketoidaan yksittäiseksi ohjelmatiedostoksi, joka täytyy lisätä kolmannen osapuolen verkkosivulle. Kokonaisuudessaan ratkaisun upottaminen vaatii 2 riviä koodia ja erittäin vähän teknistä osaamista.

Avainsanat: upottaminen, sovellus, iframe, ECMAScript, JavaScript

PREFACE

I want to thank Tuukka Toivonen for his expertise and constructive feedback. Thank you for the opportunity to have this interesting subject and your guidance in it. I would also like to thank Terhi Kilamo for her guidance on the thesis process and the mentoring received.

I would like to thank Josefiina Jolkkonen and Otto Väyrynen for their support with the thesis. Thank you for your feedback and support starting from the beginning of the thesis work. I am also grateful for the support and advise my family has given during the whole process.

Finally, I thank my colleagues at M-Files. You have given valuable expertise and support on the matter.

Tampere, 30th May 2022

Tuomas Pekkanen

CONTENTS

LIST OF PROGRAMS AND ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

DOM	Document Object Model
ES	ECMAScript, the standard behind the JavaScript language
ESR	Extended Support Release
HTML	HyperText Markup Language
IE	Internet Explorer
JS	JavaScript
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

1. INTRODUCTION

Modern web applications are more complex than they have ever been [1]. It has become common to see them around the web embedded to other web pages. Embedded content such as adverts, video players and social media widgets require little to no configuration and are relatively easy to use [2]. This is not always the case for more complex web applications. The applications may require advanced configuration to work properly and cannot always be embedded using the standard approaches.

In addition to the applications becoming more and more complex, so are the devices and contexts they are accessed from. As an example smartphones come in different sizes and operating systems, but these are only the tip of the iceberg. The web application could also be accessed from a variety of devices such as smart fridges, watches, cars or a completely custom device. On the other end of the spectrum are customers using certified environments with strict requirements. Some of these customers are still using ageing technologies such as Internet Explorer 11 even though it is retiring in June 2022 [3]. Custom fitting the application to each of these devices, their platforms and usage environments is usually not a feasible solution for the company producing the web application.

This thesis aims to solve the issue by packaging the complex web client into an easily embeddable solution. The embeddable solution could then be used by third parties to relatively easily configure the application to the devices and environments they have. By enabling third parties to extend the application usage environments the application could be made available on platforms the producing company would otherwise have no resources to support. This would benefit the company by extending the possible user base of the application.

M-Files Corporation is seeking a solution to the thesis issue. M-Files Corporation is a globally operating company which offers an enterprise content management solution called M-Files. M-Files manages enterprise content based on their metadata and enables creating workflows that support business processes. The customer organizations vary by size and the other internal tools they use. To support multiple different types of organizations M-Files is a flexible platform which is configurable to support varying use cases. One of these configuration options is integrating with existing technologies

the companies use. Examples of applications and platforms M-Files integrates with are SharePoint, Salesforce and Microsoft 365. These applications and platforms have specific integrations made by M-Files Corporation. The company wants to enable its customers to embed M-Files Web into their own content as easily as possible to further extend the configurability of M-Files.

M-Files Web client is a complex web application and needs a new method to be easily embeddable by third parties. The goal of this thesis set by M-Files is to investigate existing frameworks and to find a suitable one based on the implementation target.

The thesis uses a constructive approach to the research. The thesis creates a construct to solve the packaging of a complex web client in an easily embeddable solution. The construct is observed and based on the observations, opinions are made on how the issue should be solved. The construct is criticised based on its suitability on selected criteria and how well it fits the real-world use cases. An additional aspect considered is the theoretical contribution the construct has on the subject.

The thesis uses the created construct to answer the research questions set for the thesis. The research questions are:

- What is an efficient way to bundle a complex web application to be effortlessly used by third party developers in external sites?
- How suitable are the existing frameworks for this?

This thesis is structured as follows. Chapter 2 shows the evolution of JavaScript and the requirements it places on the development of modern applications. The chapter then introduces different embedding approaches. Chapter 3 introduces the case company M-Files, the current situation of embedding at the company and the objectives of the thesis work. In chapter 4 different alternatives for embedding technologies are researched based on the set objectives. A technology or framework is chosen to implement the embedding on. Chapter 5 shows the chosen solution and how the embedding objectives were implemented. Chapter 6 compares the result of the implementation to the set objectives and possible future development areas of the thesis work. Finally, chapter 7 concludes the thesis with a summary of the thesis work and its outcome.

2. MODERN WEB DEVELOPMENT

This chapter introduces the current state of web development from an embedding point of view. JavaScript's evolution and how it affects web development today is shown first. After that differences in browsers' JavaScript support are compared and approaches to solve the differences are discussed. Lastly, different embedding methods are presented and compared.

2.1 JavaScript Versions

JavaScript has evolved over the years since its invention in 1995. It was introduced as a "complement to Java for easy online application development" but the languages are only superficially similar in their technical designs. JavaScript has proven to be an essential part of the web, and it is still widely used over 20 years later after its invention. At the time competing browsers had to respond to Netscape's JavaScript. This response was led by Microsoft's Internet Explorer which was beginning to gain attraction. Internet Explorer 3.0 released in 1996 included JScript as a competitor to the original JavaScript due to trademark issues. [4]

As the programming languages were shattered into similar but different languages and the Internet kept rapidly growing at the time, the standardization of JavaScript was seen as a necessity. Netscape contacted the European Computer Manufacturers Association (ECMA) as a neutral party to standardize the language. Since JavaScript had a trademark the standard needed a new name which was agreed to be ECMAScript. It should be noted that nowadays ECMAScript and JavaScript are used interchangeably, even though by definition JavaScript is a specific implementation of the ECMAScript standard. [4]

Table 2.1. *ECMAScript Versions [5].*

Name	Abbreviation	Year	Notes
ECMAScript 1	ES1	1997	First standard
ECMAScript 2	ES2	1998	
ECMAScript 3	ES3	1999	
ECMAScript 4	ES4	-	Abandoned
ECMAScript 5	ES5	2009	
ECMAScript 2015	ES2015	2015	Previously known as ECMAScript 6 (ES6)
ECMAScript 2016	ES2016	2016	
⋮	⋮	⋮	Yearly releases
ECMAScript 2021	ES2021	2021	
ECMAScript Next	ES.Next	-	Dynamically refers to the upcoming version

The versions of ECMAScript are abbreviated as ES with a number following that indicates the release version. Table 2.1 lists the different ECMAScript versions. The thesis first briefly shows the early ECMAScript versions from 1 to 5. After that the more recent versions are explained in more detail.

2.1.1 ECMAScript 1-5

The first version of ECMAScript was published in 1997, referred to as ECMAScript 1 today following the standardization effort of the web [4]. The next versions followed shortly. ECMAScript 2 was released in 1998 and contained only editorial changes [5]. The last ECMAScript version of the early days, ECMAScript 3 came in 1999 and included major features still used to this date in the language: regular expressions, exception handling with the try/catch syntax and the switch command as examples [5].

ECMAScript 4 was supposed to follow soon but was ultimately abandoned [5]. The implementation of ECMAScript 4 began in early 2000 and was planned to be a major update compared to the previous releases [4]. This update led to conflict between the major companies using ECMAScript, namely Microsoft and Mozilla, the latter of which was formerly known as Netscape [4]. While Mozilla agreed on the proposed large changes, Microsoft wanted a more incremental update to the standard [4]. The standardization did not proceed as the parties could not agree on the contents of the next ECMAScript [4].

This conflict led vendors to implement their own add-ons to the standard defined features. During the time Internet Explorer had a monopoly status with 90% of the browser

market share which forced developers to adapt the add-on features. The browsers were again fragmented and developers had to implement complicated browser-specific logic to ensure the applications would work on other browsers too. Some functionalities are still available only in Internet Explorer and are contributing to the still relevant problem of needing specific logic to support Internet Explorer, which is still used in legacy applications. During the standardization halt web was still growing and was continuously evolving. JavaScript's libraries such as the still most popular JavaScript library jQuery were released. [4]

The ECMAScript standardization continued in 2009, 10 years after the release of ECMAScript 3. The next standard was ECMAScript 5 which was agreed to be an incremental update to ECMAScript 3 [4]. It was fully adopted by most browsers and included JSON parsing, array prototype methods such as map and the strict mode [4, 5]. Strict mode is a restricted variant of JavaScript which prohibits some syntaxes which are considered dangerous to use and throws errors instead of silently allowing erroneous semantics such as new global variables due to a typing error in the variable name [4, 6].

2.1.2 ECMAScript 2015

ECMAScript 2015, which was formerly known as ECMAScript 6 was released in 2015, 6 years after ECMAScript 5 [4]. From this version onward it was decided that versions are referred to by the year they are released [4]. ECMAScript 2015 is a major update and includes an extensive list of changes. These changes include the let keyword, const keyword, classes, arrow functions and JavaScript modules [6].

The let and const keywords are variable declarations meant to replace the var keyword [4]. They are block-scoped as opposed to the function scope var uses [4]. Block scope refers to creating a variable inside a code block, marked in JavaScript with curly braces. Prior to let and const keywords, up to ECMAScript 5 the var keyword was the only way to properly declare a variable. In function scope the variable is not restricted to the code block, but the function it is in or the global scope if it is not in any function [6, 7]. As an example, variables declared with var inside loops are also exposed to outside of the loop, as long as the variable is in the same function. The difference between let and const is that variables declared with const have to be assigned at initialization and cannot be reassigned afterwards [6, 8].

Classes are a concept in object-oriented programming languages that enable the creation of objects from a template [9]. Prior to ECMAScript 2015 there was no simple way to define a class. Without the concept of a class in the language, the behavior of a class can be at least partially matched by creating functions and assigning them to the class representing variable [7]. The function approach requires boilerplate code for each new function added as they have to be added to the *prototype* property of the object. EC-

MAScript 2015 introduces the *class* keyword that enables the creation of classes without having to circumvent the behavior with regular functions. The class methods are written inside of the class definition and do not require a prototype approach as was done prior to the new class syntax [7, 8]. In the upcoming ECMAScript 2022 standard private fields of classes are introduced making it possible to achieve polymorphism, inheritance and encapsulation at the same time [10].

Arrow function is a shorthand alternative to the traditional function expression. While mostly identical to the traditional function expression available in ECMAScript 5 it has some differences. For example, it cannot be used as a constructor of a class, and it does not have binding to the *this* keyword. Traditional function definitions enable binding *this* to the caller context, but arrow functions always bind *this* to the lexical scope the arrow function was defined in [11]. If an arrow function is used as a class method for example, using *this* inside the arrow function would refer to the global Window object instead of the class itself, which is usually not the desired behavior. In some cases this behavior of arrow function is beneficial, especially with methods executed at the global level: *setTimeout*, *setInterval* and *addEventListener*. With traditional functions these functions bind *this* to the global Window but an arrow function uses the context of the outer object, for example an object *setTimeout* is located in.

ECMAScript uses the module pattern for importing and exporting code [8]. These modules are encapsulated pieces of code that should be reusable and self-contained. Prior to ECMAScript 2015 the language had no built-in module functionality and the modules were created using different libraries or by simply defining the code "module" as a global variable [8, 12, 13]. The most common standards for modules in ES5 are CommonJS modules and Asynchronous Module Definition (AMD). CommonJS modules are used in the Node.js JavaScript runtime for running JavaScript as server code. The CommonJS modules satisfy the server environment requirements: they are synchronous and designed for servers [12]. AMD modules on the other hand are used on the client side, the browser. These modules are optimized for asynchronous loading and browsers [12]. Asynchronous loading is beneficial to browsers as the application load times matter and the user experience is not wanted to be blocked for loading of the modules. The different approaches are incompatible with each other. ECMAScript 2015 introduced modules as a part of the language specification, aiming to create a single module format that is compatible with all the use cases [12]. The ECMAScript 2015 modules support both synchronous and asynchronous loading, fitting to both server and client side operations [12]. They also use a static structure compared to the dynamic structure used in ECMAScript 5, enabling optimizations such as the removal of unused code during bundling and better variable checking due to them being static [12].

2.1.3 Release Process Changes

The release cycle of ECMAScript versions was changed from ECMAScript 2015 onwards to be yearly releases instead of larger and less frequent updates. The ECMAScript standardization process was also changed to be more focused on bringing single features than locking a feature set to a specific version. Brian Terlson, an editor of the ECMAScript standard commented that “Developers shouldn’t be looking at the version of the standard as much; it’s really on a feature by feature basis” [14]. The releases were changed to a 5 stage system where browsers can begin implementing a feature as soon as it is ready instead of needing to wait for the complete new ECMAScript version to be released. The stages begin from 0, an initial idea, to 4 where the feature is accepted to be included in the next version of ECMAScript [15]. Table 2.2 below summarizes the different stages.

Table 2.2. *TC39 Process Stages [15].*

Stage	Purpose
0. Strawperson	An initial idea
1. Proposal	A formal proposal with experimental implementation
2. Draft	An initial draft with minor changes possible
3. Candidate	A release candidate. Refinements allowed
4. Finished	Ready to be included in the standard

At stage 0 a member of the ECMA TC39 committee introduces a new change idea to the specification. At stage 1 the benefits of the change are explained, API changes are considered at a high level and potential challenges are identified. Stage 2 requires the change to have precise semantics and syntax described. To move to the candidate stage the proposal change is close to being final. Stage 4 features become an official part of the next ECMAScript version to be released, *ES.Next*. Based on the different stages browsers can begin implementing the changes before the official new standard version is released. Proposals at stage 4 are considered stable and proposals at stage 3 are not expected to receive any major changes. The benefit of integrating the proposals earlier than at the next official release is more gradual updates. [15]

2.2 JavaScript Support In Browsers

There are multiple widely used browsers that can access websites and the web applications in them. The way these browsers interpret the application code is not the same. There are differences between browsers from different publishers but also between different versions of the same browser [16]. To compare the differences multiple browsers are chosen from different publishers for comparison. The browsers selected are Google Chrome, Mozilla Firefox, Safari (Apple), Microsoft Edge, Microsoft Internet Explorer and

Samsung Internet Browser. With the exception of Samsung Internet, all of the browsers are available on a desktop environment. Samsung Internet is selected to present mobile only browsers. Safari is available for both desktop and mobile and both versions have identical support of ECMAScript. From each browser 3-4 versions are chosen. One of these versions is a currently supported, recent version of the browser. In addition to the modern version 2-3 older versions of the browser are selected. All of these older versions are not officially supported anymore, but still retain active users as of February 2022.

Table 2.3 compares the support of ECMAScript for these browsers. Each browser version can support a different amount of the features introduced in a version of ECMAScript. The differences come from the different JavaScript engines used in the browsers. The browsers are compared based on how many of the features they support of a specific ECMAScript edition, which is then converted to a percentage. The ECMAScript versions of the browsers are compared against are ECMAScript 5, ECMAScript 2015, ECMAScript 2016 to 2022 as a bulk and the upcoming features in ECMAScript Next. In addition to the ECMAScript support a market share of the chosen browser is provided. The market share should be considered only indicative, since the release cycles of the browsers differ and users update to different versions at different times. Especially on the older versions there are anomalies where larger portions of users are active. Chrome for Android is the leading browser by market share for mobile devices running Android but was not included due to scarce data of its different versions. It should be noted that Edge has moved to use Chromium starting from Edge 79. Chromium is the open source base of Chrome. With the version numbering of Chrome and Edge following the Chromium releases, Chrome 87 and Edge 87 use the same JavaScript engine and have identical support of ECMAScript.

From table 2.3 several observations can be made. Starting with the market shares, the main thing to note from market shares is that there is still a substantial user base using older browsers. Though the numbers are individually a lot lower than the currently supported counterparts, it should be noted that there are multiple versions between these older versions with somewhat similar market shares. This user base using older versions of browsers may encounter issues with applications if they are not optimized to support older browser versions. Most of these old versions should not be considered ancient either, for example the oldest shown Firefox 78 Extended Support Release (ESR) was released on June 30 2020, and is the newest supported version in Apple's macOS versions 10.9, 10.10 and 10.11 [17]. An anomaly in the market share data of old browsers is Internet Explorer. Especially Internet Explorer 11 is still widely in use, more so than the still supported Firefox 91 ESR or Safari 14 for example. Internet Explorer 8-10 also still have active users though much fewer than on Internet Explorer 11. An anomaly on the other spectrum is the currently supported Chrome 98 holding a comparatively massive market share of 15.53%, over 7 times the amount of the currently most active Firefox version 97.

Table 2.3. *ECMAScript Support of Different Browsers [3, 16].*

Browser \ ECMAScript	ES5 (%)	ES2015 (%)	ES2016+ (%)	ES.Next (%)	Market Share (%)
Chrome 87	100	98	92	7	0.13
Chrome 91	100	98	94	7	0.11
Chrome 98	100	98	100	12	15.53
Firefox 78 ESR	98	98	76	7	0.06
Firefox 91 ESR	100	98	96	7	0.10
Firefox 97	100	98	99	7	2.03
Safari 13	100	99	59	4	0.05
Safari 14	100	99	71	7	0.24
Safari 15	100	99	90	7	0.15
Edge 18	100	96	28	4	0.07
Edge 87	100	98	92	7	0.01
Edge 95	100	98	100	7	0.03
Internet Explorer 8	12	-	-	-	0.04
Internet Explorer 9	82	-	-	-	0.06
Internet Explorer 10	97	3	-	-	0.03
Internet Explorer 11	97	11	1	-	0.62
Samsung Internet 12	100	98	72	7	0.03
Samsung Internet 13	100	98	78	7	0.10
Samsung Internet 14	100	98	92	7	0.11

It has become apparent that older browser versions still have a significant amount of users and supporting them should be considered. Next are observations from the ECMAScript support of the different browsers. A support of 100% means a developer can assume the browser to perfectly support all the features in the ECMAScript edition. A support of over 90% in turn means the edition is almost fully supported but a few specific features can be missing. Values under 90% mean that issues with features from the ECMAScript edition become more and more common all the way to 0% where none of the features are functional.

The general trend with ECMAScript 5 (ES5) is that it is almost completely supported,

with a few exceptions. Firefox 78 ESR is missing support for proper exponential number rounding. Internet Explorer 10 and 11 are missing the same exponent rounding and in addition one miscellaneous feature. Internet Explorer 8 supports almost none of the features and 9 supports a usable amount of 82%. For the next editions of ECMAScript Internet Explorer is practically unusable with support almost completely missing. Internet Explorer is excluded from the next comparison notes when all browsers are mentioned.

ECMAScript 2015 (ES2015) is a major update from ES5, but browsers have had time to implement the changes by now. Across the board the features supported are very close to the full support. Except for Safari, the browsers are all missing tail call optimization used in recursion. Edge 18 is missing multiple implementation details of features, but still partially supports all features except for the tail call optimization.

ECMAScript 2016 to ECMAScript 2022 (ES2016+) has differences in support between browsers and individual browser versions. Chrome and Firefox support the features well even across older versions, with Firefox 78 ESR being the exception with a support of 76%. Safari has worse support for ECMAScript 2016 - 2022 than Chrome and Firefox. By release date Safari 14 is comparable to Firefox 78 and supports 5% less of the features. Even the currently supported Safari 15 supports 90% of the features while the earlier released Firefox 91 ESR supports 96%. Chrome 91 released 4 months before Safari 15 supports 94% of the features. If supporting Safari is planned, its lack of similar ECMAScript support to Chrome and Firefox should be paid attention to. Edge has identical support to Chrome starting from Edge 79. Edge 18 is the last version before the transition to Chromium base and has a noticeable spike in user count. Note that the most used Edge version currently is Edge 98 with a market share of 2.98%, but it does not have enough data for ECMAScript support comparison. Edge 18 only supports 28% of the ES2016+ features. Similar to Safari, Edge support needs special attention if planning to capture the users still using Edge 18.

Samsung Internet is the only browser selected that is exclusively on mobile devices. On ES2016+ it lacks behind other browsers released during the same time. Samsung Internet 13 was released the same month as Chrome 87, but their coverage of ES2016+ has a difference of 14%. Samsung Internet lacking behind Chrome is not due to using a different JavaScript engine, it is based on the same Chromium browser as Chrome is. The difference comes from the delay Samsung Internet has on updating the Chromium base it has. Samsung Internet 13 is based on Chromium 79, while Chrome 87 released at the same time is already using Chromium 87. Samsung Internet 14 updated to Chromium 87 and has identical results to Chrome 87, but was released half a year later. Supporting Samsung Internet needs consideration for the infrequent Chromium base updates it has.

Overall browsers can be seen to catch up with new features introduced, but the process is not instantaneous. New features are arriving each year with a new ECMAScript release

which browsers need to support. ES.Next contains the upcoming features of stages 2 and 3 that are considered for the next official standard edition. The support of these is minimal from all browsers, but this is expected since these features are not officially accepted to be included in the next standard yet.

As solutions to the fragmentation of the ECMAScript support in different browsers, transpilers and polyfills are introduced.

2.3 Transpiling

Transpilers, or transformation compilers, are a subset of compilers that perform source-to-source translations [18]. The transpilation can be from one programming language to another one or to the same programming language with modifications compared to the original source code [18]. A transpiler can be used for example to translate from the Python programming language to JavaScript or from a newer version of the ECMAScript standard JavaScript to an older version [8, 19]. The transpiler is usually integrated as part of the build process for automation.

For the purposes of this thesis the above-mentioned JavaScript to JavaScript translation is needed. As noted in 2.2 all browsers do not support the newest ECMAScript versions. With transpilation it is possible to use the syntax from newer versions and still run the code on older browsers having outdated ECMAScript versions in use. While it is possible to use vanilla JavaScript directly when typing the source code, it would lead to missing out on productivity through the new syntax added to newer versions of the language. Instead of waiting for the standards to finalize it is recommended to use the newest version and transpile it back to an older version [8].

An important aspect to note is that transpiling only allows using newer syntax in the case of transpiling JavaScript to JavaScript. The syntax is transpiled to the older version making it compatible with older browsers.

Program 2.1 is written with the ECMAScript 2015 standard. In particular, it uses the let scope syntax and arrow function definitions. This and all the following transpiled programs are kept as faithful as possible to the transpiler generated code. Comments are added and line breaks modified for readability where needed but the code is otherwise kept intact.

```
1  "use strict";
2
3  // Let syntax variable initialization
4  let author = "Tuomas Pekkanen";
5  let pages = 5;
6
7  // Const syntax variable initialization
8  const book = {
9      // Shorthand property names
10     author,
11     pages,
12     // Function defined with arrow syntax
13     printAuthor: () => console.log(
14         // String substitution ($) used to inject variable to text
15         `The book was written by ${author}.`)
16 };
17
18 book.printAuthor();
```

Program 2.1. *Chrome 92 with ES2015 compatible JavaScript snippet Book.js.*

The code initializes variables *author* and *pages* on lines 4-5. The variables are initialized with the let syntax which is block-scoped. The variables are then used to initialize a book object. The book object has an additional function *printAuthor* defined on line 13 using the arrow syntax notation that prints the author of the book. The author is substituted to the string using a template literal marked with the $ symbol.

The same Book.js script can be transpiled to support older browser versions which have not implemented the ES6 standard, but are using the older ES5 version. The same source code is transpiled targeting Chrome 40 which supports only up to ES5 below. The syntax has changed but the transpiler keeps the behavior identical to the original source code. The transpilation is done with Babel. The transpiled version is shown in Program 2.2.

```
1  "use strict";
2
3  // Var syntax variable initialization
4  var author = "Tuomas Pekkanen";
5  var pages = 5;
6
7  // Var syntax variable initialization
8  var book = {
9      // Full property declaration
10     author: author,
11     pages: pages,
12     // Full function definition
13     printAuthor: function printAuthor() {
14         return console.log(
15             // String concatenation used to inject variable to text
16             "The book was written by ".concat(author, "."));
17     }
18 };
19
20 book.printAuthor();
```

Program 2.2. *Chrome 40 with ES5 compatible JavaScript snippet Book.js.*

The variable and object declarations have changed to use the older var syntax which creates globally scoped variables instead of the block scope let uses. The function *printAuthor* has changed to the formal definition containing the function keyword, return statement and curly braces. The template literal that substituted the author to the printed string has changed to a string concat function which appends strings.

A more complex example is using the class syntax introduced in ES2015. This syntax allows creating class-like objects in JavaScript similar to object-oriented programming. Program 2.3 demonstrates the class syntax of ES2015.

```
1  // Define a class
2  class Library {
3    // Class constructor
4    constructor(name) {
5      this.name = name;
6    }
7
8    // Getter function
9    get name() {
10     return this.name;
11   }
12 }
```

Program 2.3. *Chrome 92 with ES2015 compatible JavaScript snippet Library.js.*

The class defines a library which contains the name of the library. The class' properties defined in the constructor are implicitly public. To get the name of the library a getter function is introduced.

Program 2.3 must be transpiled to support browsers without ES2015. The program is transpiled to support ES5 in Program 2.4. The class syntax does not exist in ES5 and is circumvented using the old syntax.

The program 2.4 may be more understandable when following the execution order rather than reading from top to bottom. The class implementation is completely different due to the transpiling. Starting at line 27 the library is defined as a function in absence of the class keyword. The function defines an inner function at line 29 similar to the class constructor in Program 2.3. It has in addition a call to *_classCallCheck* which verifies the class is called only through objects and not the class itself. With the constructor ready the class functions are added to it starting from line 34 with the *_createClass*. The function adds the *name* property to the constructor object. In *_createClass*, with some simplifications, the properties are just looped through and added to the constructor object. The ready constructor object is then returned as it contains all class members and functions. The class can be used exactly like in program 2.3 even though the implementation of it has changed.

```
1  // Prevent calling the class itself as a function
2  function _classCallCheck(instance, Constructor) {
3      if (!(instance instanceof Constructor))
4          throw new TypeError("Cannot call a class as a function");
5  }
6  // Add properties to the class.
7  function _defineProperties(target, props) {
8      for (var i = 0; i < props.length; i++) {
9          var descriptor = props[i];
10         descriptor.enumerable = descriptor.enumerable || false;
11         descriptor.configurable = true;
12         if ("value" in descriptor) descriptor.writable = true;
13         Object.defineProperty(target, descriptor.key, descriptor)
14     }
15 }
16 // Create the class by adding properties to the constructor
17 function _createClass(Constructor, protoProps, staticProps) {
18     if (protoProps)
19         _defineProperties(Constructor.prototype, protoProps);
20     if (staticProps)
21         _defineProperties(Constructor, staticProps);
22     Object.defineProperty(
23         Constructor, "prototype", {writable: false});
24     return Constructor;
25 }
26 // Define the Library class as a function
27 var Library = /*#__PURE__*/function () {
28   // Class constructor mimic
29   function Library(name) {
30     _classCallCheck(this, Library);
31     this.name = name;
32   }
33   // Add the properties to the constructor.
34   _createClass(Library, [{
35     key: "name", get: function get() { return this.name }}
36   ]);
37   return Library;  // Return with class members and functions
38 }();
```

Program 2.4. *Chrome 40 with ES5 compatible JavaScript snippet LibraryES5.js.*

Transpilation has also made it possible to create new extended programming languages that are based on the language they transpile to. TypeScript is JavaScript with syntax checking added to it, a superset of JavaScript [20]. The type checking enables tighter integration with the editor for error checking and can be considered safer than JavaScript for avoiding invalid type errors. The downside is needing to use types everywhere is the time it takes to add them, which can decrease productivity. TypeScript is transpiled to JavaScript before the browser interprets it.

Transpilers are a relatively new use case of compilers and are continuously evolving. A recent study in the field has shown that transpiling Python to Rust as an intermediate step before compiling to machine code yields performance gains up to an order of magnitude while using less memory on a conventional desktop computer [21].

2.4 Polyfilling

Polyfills are code pieces that enable modern technologies to be used on older versions of browsers. While transpiling is related to syntax, polyfills bring new features to the API. Another difference to transpiling is that polyfills do not require compilation, or in this case transpilation, to implement them. They are used when the syntax can remain the same but the implementation is missing in older browser versions. [8]

The inventor of the polyfill term Remy Sharp describes that the term polyfill is a more general concept than just for JavaScript usage when explaining how he came up with the term: “Polyfill just kind of came to me, but it fitted my requirements. Poly meaning it could be solved using any number of techniques - it wasn’t limited to just being done using JavaScript, and fill would fill the hole in the browser where the technology needed to be. It also didn’t imply "old browser" (because we need to polyfill new browser too).” [22]

Browsers natively support different APIs as introduced in 2.2. Newer versions usually have more APIs available than their older versions. From the MDN Web Docs it can be seen that Internet Explorer 11 does not have support for many of the String.prototype methods while almost all modern browsers support these [23]. If the developers still want to use methods from String.prototype such as *startsWith* and support Internet Explorer 11 a polyfill can be used as shown on the program 2.5 below. The function *startsWith* checks whether a string begins with the parameter string. It has one required parameter search, the string that is checked for at the beginning, and an optional parameter pos, the search position which defaults to zero. The position is used to set a positive offset to the search from the original string, for example a value of three would mean that the search string is looked for beginning at the third position.

```
1 if (!String.prototype.startsWith) {
2     String.prototype.startsWith = function(search, pos = 0) {
3         return this.substring(
4             pos, search.length + pos) === search
5     }
6 }
```

Program 2.5. *Polyfill of String.prototype.startsWith.*

The program 2.5 has the common structure of a polyfill. First an API guard is included on line 1 to only use the polyfilled version if the native version does not exist. The second part is defining the missing function as seen on line 2, after which follows its implementation according to the modern specification of the function. The implementation uses the string substring function which exists in most older browser versions.

While polyfills are needed to support older browsers they might have a cost associated with them in the form of performance. More polyfills included means more code to load which is part of the performance slowdown. Another reason is that the native browser API is usually more performant than a polyfill of the same feature [24]. If the browser already has the feature implemented the polyfill can be considered an additional overhead which is why the API guard is an important part of a polyfill. Even with an API guard the code still has to be loaded leading back to the first reason of a polyfill's performance cost.

Different browsers and their different versions support different APIs as shown in 2.2. As the API is fractured among the different browsers and their versions, so are the polyfills needed to support them. Supporting multiple browsers and browser versions requires attention to what polyfills are provided to them from the application, since the same polyfill may not be a fit even within the same browser if the version of the browser is different.

2.5 Embedding Content

Embedding means incorporating content from another source within the body of a web page [25]. The embedded content is used for various different purposes in web pages. At its simplest embedding can be adding an image to a website. The image is loaded from an external source, a file on the machine or from another website through a Uniform Resource Locator (URL) [26]. This approach uses the HTML *<img>* tag. As the actual image resides outside of the web page it is considered an embedded element.

In HTML5 similar embedding structures can be used to embed video or audio content, the *<video>* and *<audio>* elements [27]. Prior to HTML5 video and audio content were commonly embedded using plugins that had the capability to show those content types. The plugins are mostly replaced by HTML5 because of security issues [27].

2.5.1 Iframe

Iframe, or inline frame element, is an HTML element that contains a web page and embeds it to the current page [28]. The difference to media elements like *<img>* and *<video>* is that there is no direct control over the element provided. With a *<video>* element the video is controlled and held by the web page creator, but this is usually not the case with an iframe. Iframe is used to embed third-party content into the website that the host of the page does not have direct control over [27]. An iframe can be used for example to embed a video including the video player from a third-party video service provider [27]. With a *<video>* element the video should be prepared in multiple different video formats and a video player has to be implemented by the host website when following best practices. Failure to do so may lead to the video not being playable with different browsers that do not support the video format or the video not having volume controls at all if not implemented in the player. With an iframe the third party is responsible for offering the whole content, including the video formats and video player in the case of a video [27]. In addition to the media elements, iframe has other more complex use cases. As it can be used to embed practically anything from the web its usage has also expanded to different areas, such as interactable map applications and complex web applications.

Configurability is an important aspect when the embedded content gets more complex. An image can be configured with a single URL to the image resource along with its width and height. A video player requires width, height and a source too, but in addition it possibly needs a timestamp where the video is started from and caption settings as some examples. A map application similarly can be configured to zoom to a specific location and the map layer can be specified as well. These configurations have to be transferred to the iframe in some way. With the video player and map examples all the configuration data itself is not sensitive information and can be passed on via the source URL of the embedded content itself. In the example iframe element in program 2.6 a map application is embedded.

```
1 <iframe width="425" height="350"
2     frameborder="0" scrolling="no"
3     marginheight="0" marginwidth="0"
4     src="https://www.openstreetmap.org/export/embed.html?
5         bbox=23.85%2C61.44%2C23.87%2C61.45&amp;layer=mapnik"
6     style="border: 1px solid black">
7 </iframe>
```

Program 2.6. *Iframe element embedding content from OpenStreetMap.*

The iframe contains the default attributes for the element itself: *width*, *height*, *frameborder*, *scrolling*, *marginheight* and *marginwidth*. These control the behavior of the iframe

element, mostly focusing on how the iframe is positioned on the site [28]. The configuration sent to the application itself is in the definition of the *src* attribute. The URL contains URL query parameters *bbox* and *layer*. The *bbox* contains the location the map opens to and the *layer* the map layer, in this case the default view mode of the map application. The URL can be opened independent of the rest of the iframe element and the application would still point to the same location.

While URL query parameters are useful in these cases it has a severe downside. The URL parameters can be examined by anyone from the website's source code. Using URL parameters to send login information or any other sensitive information is not safe. The communication is also only allowed in one direction, from the host to the iframe and not the other way around with query parameters. The application would have no way to send information back to the host site. These features might be necessary for a complex web application and other ways of transferring the configuration and communicating with the iframe are needed.

A simple way of communicating with the iframe is directly manipulating it with JavaScript. The iframe element is accessible directly with JavaScript and its contents are modifiable, but only if the iframe is from the same origin [28]. The same-origin policy is a security feature that places restrictions on interaction between the origin and the external source. Two sites are considered to be of the same origin if they have the same protocol, domain and port [29]. The policy for example prevents an iframe from accessing most of the content of the parent site to prevent potentially malicious behavior such as reading all data from the parent window and sending it to an external actor [28, 29]. In addition to same-origin policy the iframe is recommended to be run in a sandbox environment which restricts many functionalities considered as a security risk or otherwise malicious behavior: downloading files, opening popups and executing scripts as some examples [28, 29]. If communication is wanted to be done by direct manipulation the sandbox would need to be configured to allow script execution at the least. Finally, if the website is using the encrypted HTTPS protocol instead of the HTTP protocol, the embedded content has no access to the parent site at all and the parent site has no access to the embedded content due to the encryption [27]. All in all, while it is possible to directly manipulate the iframe content with JavaScript it has limitations due to security reasons. If the limitations are not an issue accessing the iframe can be done by getting the iframe element and then modifying its properties. An example HTML document with a script to get content from an iframe is shown below in program 2.7.

```
1  <iframe id="iframe" src="http://example.com" </iframe>
2
3  <script>
4      // Get the iframe
5      let iframe = document.getElementById("iframe")
6      iframe.onload = () => {
7          // Get the heading from the iframe document
8          let iframeDocument = iframe.contentWindow.document;
9          let title = iframeDocument.getElementsByTagName("h1")[0];
10         // Display the heading in an alert on the parent document
11         alert(title.textContent);
12     }
13 </script>
```

Program 2.7. *Accessing the heading of an iframe document from the parent.*

The program 2.7 begins with the *iframe* element which embeds a simple example site to the host site. The example site has a heading element in it that the host site wants access to in this example code snippet. The *script* element contains JavaScript code which handles getting content from the embedded site. On line 5 the *iframe* element introduced on line 1 is acquired from the document. After the *iframe* has finished loading the content it displays the code begins manipulating it. Line 8 stores the HTML content of the iframe to the variable *iframeDocument*. From this variable the title is then stored to a variable on line 9 by searching for the heading tag *h1*. Finally, the code displays a window on the host site that shows the heading from the embedded site. The code could also modify the contents of the heading or execute a script in the embedded page with minor modifications. The example is functional only if the example site is from the same origin as the site embedding it. If this was not the case a browser dependent error message would be displayed regarding the same-origin policy violation.

The third method of communicating with an iframe is the cross-document messaging API of Window objects. A window object can refer to a page, pop-up or an iframe embedded into another page [30]. The previous communication methods have limitations from the cross-origin security measures making their communication with an iframe limited. The *Window.postMessage* method is created for the purpose of safely sending messages to a window of another origin. Using the postMessage it is possible to send an event object containing configuration instructions or other communication such as instruction to call a function to the foreign origin. The communication is not limited to interaction from parent to iframe as the iframe can use the same methods to communicate to a parent window. The recipient of the message has to explicitly register itself to the event messages from the window object to receive data from another origin. The recipient can

verify the sender of the message and choose whether to do anything or not [30]. This way the messaging can occur in a more controlled way as the sender of the message does not need permissions to freely execute scripts on the receiver risking security, and the receiver has control over how to react to a message.

To send a message the sender must obtain a reference to the window the message is being sent to. If the receiver is an iframe the window can be obtained similarly to Program 2.7 on line 10. The difference being only needing to store the *contentWindow* instead of the *document*. If an iframe is to communicate with the parent window it can get a reference to it from the window object hierarchy using *window.top* if the target is the topmost window in the hierarchy [30]. Using this reference the *postMessage* method can be used. The method requires 2 parameters: *message* and *targetOrigin*. *Message* contains the data to send to the target window. The data passed to it is automatically serialized with the structured clone algorithm which copies complex JavaScript objects safely removing the need for the user to serialize the data [30]. The *targetOrigin* parameter is a security measure that limits where the event can be dispatched to. It can be either * for any target or a URI of the target window. When the event is to be dispatched the scheme, hostname and port of the referenced window are checked against the *targetOrigin* parameter. If they all match the message is dispatched, otherwise the message will not be sent. This prevents a malicious actor from intercepting the message which may contain sensitive data. Mozilla recommends to always specify the *targetOrigin* if it is known [30]. The Program 2.8 below shows an example of sending a message to an iframe.

```
1  <iframe id="iframe" src="http://example.com" </iframe>
2
3  <script>
4      // Get the iframe
5      let iframe = document.getElementById("iframe")
6      // Get the iframe window object
7      let iframeWindow = iframe.contentWindow;
8      // Send configuration to the iframe
9      let configuration = { active: true };
10     iframeWindow.postMessage(
11         JSON.stringify(configuration), "http://example.com");
12 </script>
```

Program 2.8. *Sending a message to an iframe of different origin.*

The beginning of the program is similar to the Program 2.7 with differences in storing only the window itself and not needing to wait for the document to load as it is not referenced. The program sends a configuration to the iframe with a boolean variable *active* set to true. The configuration is parsed to a string with *JSON.stringify* on line 11. The manual

parsing is not required with modern browsers, but Internet Explorer 8 and 9 do not support objects passed in the message, only raw string data. The JSON parsing is only required if support for Internet Explorer 8 and 9 is planned. Internet Explorer 6 and 7 do not have support for the *postMessage* at all [30]. The second parameter is the *targetOrigin*, the origin of the iframe source. Only *http://example.com* is able to receive the message, or to be precise, it is not dispatched at all from the sender if the URI does not match to the *iframeWindow's*.

To receive messages the receiver has to subscribe to the events by adding an event listener. The event listener is added to the window object of the receiver using *window.addEventListener*. The event listener has to specify the type of events it will react to in the first parameter. In this case the type is *message* as events from window objects from different contexts are classified to the MessageEvent type. The second parameter is a listener object which receives the event object. The event object from a *postMessage* event will contain a data, origin and source properties [30]. The *data* contains the serialized message from the sender. *Origin* is the origin of the window that sent the message. The origin is the origin at the time of sending the message even if the sender navigates to another origin before the receiver gets the message. Program 2.9 below adds an event listener to the messages from program 2.8.

```
1  // Configuration of the application.
2  let active = false;
3
4  // Listen to messages from windows with different origin.
5  window.addEventListener("message", (event) => {
6      // Verify the origin of the sender.
7      if( event.origin !== "http://mysite.example" )
8          return;
9
10     // Get and verify the message.
11     let messageObject;
12     try {
13         // Parse the message back to JSON.
14         messageObject = JSON.parse(event.data);
15     } catch(error) {
16         // Handle invalid JSON syntax.
17         return;
18     }
19     // Verify the 'active' property.
20     if(!messageObject.hasOwnProperty("active") ||
21         typeof messageObject.active !== "boolean") {
22             return;
23     }
24
25     // Set the configuration as received from sender.
26     active = messageObject.active;
27
28     // Respond to the sender.
29     event.source.postMessage("OK", event.origin);
30 });
```

Program 2.9. *Receiving a message in an iframe from a different origin.*

The program 2.9 is implemented in the receiver side, http://example.com. It receives the configuration as an event message and stores it to the variable *active*. To receive the message the program subscribes to events of *message* type on line 5 and reacts to a message with the function defined below it. The function receives the event data in the parameter *event*. The event contains the origin of the sender which is checked on line 7. The program assumes the sender to have origin http://mysite.example and if it does not, the message is rejected. This prevents potentially malicious third parties to send messages to the program. On lines 12-23 the contents of the message are checked. The

message sent from program 2.8 is in the *data* property of *event*. While the site itself is at this point of the program trusted, the contents of the message it sends should not be automatically trusted. If the trusted sender has a security breach it could send malicious data [30]. The *data* is parsed to JSON on line 14. If the conversion fails, meaning that the message is not valid JSON, the message is rejected in error handling on line 17. From line 19 onwards the message is valid JSON and has been parsed. Lines 20-23 make sure the message has the configuration property *active* in it and that the type of it is boolean. As JavaScript is weakly typed the sender could send a completely different type as *active*, for example a Date, causing problems if accepted. With the validation done the program accepts the sent message and assigns the *active* state to itself on line 26. The program can send a message back to the sender by using the *source* property of the *event* parameter and by using the sender origin as the new receiver origin. Program 2.8 does not implement a receiver for messages and would not receive the message, but an event listener could be added to it similarly.

2.5.2 Script-Based Embedding

Vinegar et al. presents script-based embedding, embedding with iframes and challenges in using them [2]. Script-based embedding is a more dynamic approach to embedding content. Using JavaScript it is possible to dynamically modify HTML content of the host page to include embedded content. One way of embedding content with JavaScript is using the *<script>* tag and including an external script source to load with it. The external script can then dynamically create new HTML elements to the page, holding the content it embeds to the page.

A commonly seen example of this dynamic content are targeted adverts on websites. These adverts show content based on data gathered from the user to show adverts that are more likely to be relevant to the user. This type of content is dynamically loaded by a script included on the page from an advertisement provider. Because the content is created on the fly it is possible to determine and show a specific advert to the user instead of a generic advert. Other examples include small applications included on the page, widgets. The driving factor of these widgets is that they are externally hosted and typically easy to embed to the page. Disqus is an example of such widget. It is a commenting widget which can be embedded to a page with a few lines of code and enables adding an embedded section to the page where users can comment.

When comparing script-based embedding to iframe several benefits can be found, but not without some disadvantages. Communicating with script generated content is directly possible. If the content is included in an iframe, the embedded content cannot be directly modified and the embedded content cannot access the parent site if embedded from an external source. This brings security to the iframe approach at the cost of configurability.

The script generated element can be flexibly modified, but it also has unrestricted access to the content of the host page. The modifiability enables a more custom integration of the embedded content. The script-based content can be dynamically resized as needed while an iframe stays the same size it is initially loaded in. The script-based content can inherit styling with CSS from the parent site and look coherent with the parent site. The iframe content uses the style of the external content provider and does not inherit the CSS from the parent site. Another benefit to the script method is that it is also possible to gather analytics from the application usage, such as the time users spend viewing the application.

As the content embedded with JavaScript is under control of the website host, there are additional challenges when developing the application to be embedded. The first challenge is the unknown context of the host page. The host has full control of the DOM layout and where the application is embedded to. The host can for example embed the application to the *<head>* tag at the top or to the end of the page at the bottom of the *<body>* tag. In HTML5 it is also possible to omit the *<head>* tag completely. If the application depends on being in a specific location or expects the *<head>* tag to exist it may cause issues. The host browser can also interpret the HTML differently based on the mode it is running in. If the site is made to support Internet Explorer 5 for example it will run in quirks mode which emulates the nonstandard behavior required for Internet Explorer 5 [31]. The browser may also turn to quirks mode if the HTML of the application is malformed.

The second challenge is the shared environment of the page. When the application is embedded with a script it shares the one and only global variable namespace with the rest of the page. This shared global namespace leads to two possible issues. First, the embedded application should not pollute the namespace with its own global variables. Second, the global variables in the namespace may be modified by other applications or the host page itself. For example if the application depends on JSON parsing from the global JSON object the possibility of it being modified has to be considered. Especially on older browsers custom JSON parsing implementations are used as the native parsing may not exist. Some of these implementations are not compatible with the standard JSON parsing and may yield different results for the same inputs. Embedded application developers should consider the risks of using the global namespace to avoid issues from incompatible implementations. Similar caution should be used if the application modifies or depends on the DOM tree. There is only one DOM that is global to the whole page and includes possible third-party applications in it. If the application inserts new elements they have to comply with other applications running on the page and not interfere with them. Other applications can query the DOM, so unique element IDs and class names should be used. The same issue applies backwards: the application can accidentally select additional or wrong elements from the DOM if the selectors are in collision with

other applications on the page. [2]

Lastly, the third challenge of browser restrictions. Similarly to the iframe, the script-based solution has to address the same-origin policy issue. The application cannot access origins other than the one it is currently running on without complying to the cross-origin resource sharing (CORS) [2, 32]. Another limitation comes from third-party cookie restrictions. The browser can, depending on the settings, restrict reading or writing third-party cookies. This in turn may prevent the application from logging the user in if the session information is stored to the cookies [2].

2.5.3 Embedding Frameworks

Embedding frameworks in this thesis refer to solutions that enable embedding content into the framework provider's platform. Platform providers may define the embedded applications under different names, such as *components* or *add-ins* to their platform. The frameworks allow the creation of application bundles that the provider platform then hosts. The frameworks offer additional tools for integrating the embedded application to the platform. As an example, the configuration of the embedded application could be done via a configuration pane provided by the platform. The configuration pane is native to the host platform and the behavior is in general the same for all applications embedded to the platform. The tools provided by the platform may enable deeper integration with the platform, but the downside is the tooling being platform-specific. Since the frameworks are provided by different commercial operators, switching between platforms or adding support to a new framework requires additional work. At a general level, the frameworks are based on dynamic content creation like the script-based embedding. The difference is that the framework offers tooling to embed content into their platform, while with script-based embedding a platform provides a tool for embedding their content into another host. Each embedding platform is different, but the basic principles of them discussed here are common. For context 2 major embedding frameworks are introduced: Salesforce and SharePoint.

Salesforce is a customer relationship management (CRM) platform [33]. A CRM platform focuses on managing the relationships and interactions with existing and potential new customers [34]. Salesforce allows extending its functionalities with applications from external developers. These applications are made with their Lightning Component framework. The framework is a UI framework for making single page applications, but also allows using HTML and JavaScript for the creation of the application. The framework supports ECMAScript 5 syntax and additionally Promises from ECMAScript 2015 [35]. The Salesforce platform is entirely cloud-based.

Microsoft SharePoint is a platform for creating websites. Its main features include content collaboration and management. Sharepoint can be configured as a fully hosted cloud

service or as an on-premises installation [36]. SharePoint has "modern pages" where "web parts" can be added to customize the page. The web parts can contain text, images, videos or dynamic content as some examples [37]. SharePoint allows external parties to create add-ins that are embedded to the SharePoint pages. The add-ins can be hosted by SharePoint or the add-in provider. To create a web application, the self-hosted add-in is the only allowed way. The application can be made with any technology stack. SharePoint offers tooling for customizing the application to look like other SharePoint applications. In addition, the add-ins can use SharePoint API to integrate with SharePoint's features [38].

3. CASE M-FILES

M-Files Corporation is a software development company based in Finland. M-Files has over 500 employees worldwide as of 2022. The company has offices in Finland, United States, United Kingdom, Sweden, Germany, France, Canada and Australia. In Finland the offices are in Tampere, Espoo and Lappeenranta with Tampere office being the corporate headquarters. [39]

M-Files operates globally in over 100 countries and has thousands of organizations as customers. The customers of M-Files are typically from consulting, law or industrial background. M-Files was founded in 1987 and began the development of the information management product which is also called M-Files in 2002. [39]

3.1 M-Files Product

M-Files Corporation's primary product is the enterprise content management solution M-Files. The M-Files product is used to manage the large volumes of content enterprises might have. This content can be for example: documents, process workflows, employee data and customer data. The content is structured in M-Files by the metadata of the content instead of the folder structure and name of the file. Metadata means information of the data itself. A project document for some company may include as metadata the customer company, the date and price of the project for example. The metadata can be gathered from the content inserted into M-Files or manually edited by the user. The content stored in M-Files is not put in a folder like a traditional file explorer. Instead, the user can search for the content based on the metadata. In other words, the user does not have to remember the location or names of the files, they can directly search for the project company or any other metadata to find the project document. M-Files can be accessed using a Windows desktop application, a smartphone application or from a web browser as seen in figure 3.1. In addition to these an API is available for headless access to the software. This thesis focuses on the web client used to access M-Files. [39]

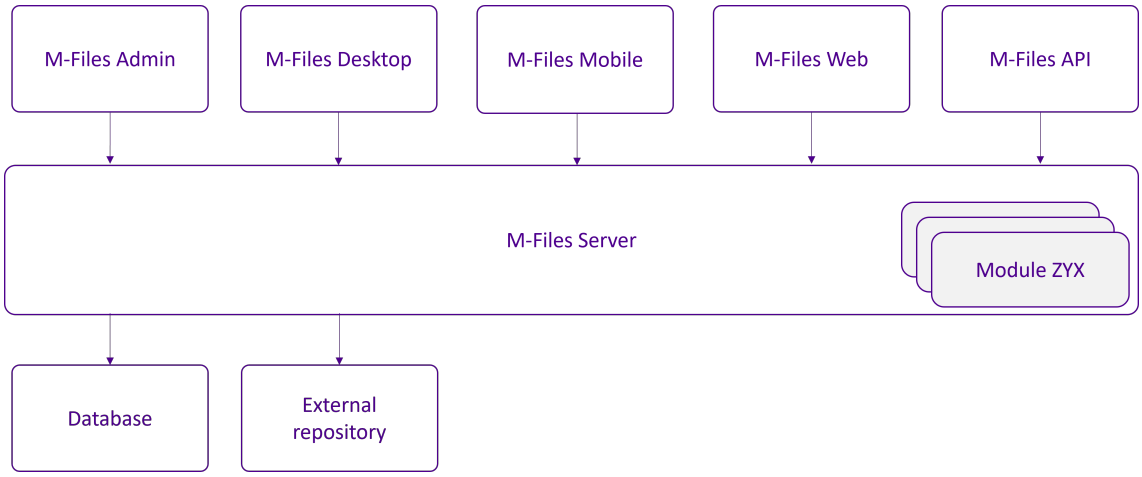

Figure 3.1. *An overview of the M-Files product architecture. Adapted from [40].*

An object in M-Files is a container of information. It has metadata and can contain any amount of documents in it. All objects have an object type, which defines how the objects behave. One object type is *document*, but it can as well be a *customer* or a *project*. A *document* requires a file to be attached to it while a *customer* may require only metadata of the customer it represents. The metadata itself is added to the object as key-value pairs that are called properties of the object. The properties available depend on the object type and the more detailed class of the object type. Some properties are mandatory. Some are automatically gathered from the object data and not editable, such as the version and modification date. Objects can be related to other objects through relationships. They can have their visibility and editability controlled by named access control lists. An object may have a workflow which is related to the customers' real-world processes to manage how the object transitions between different process states. [39]

A user is able to search for objects based on their metadata or the contents of the documents it may contain. The objects are not grouped to folders, unless specifically done so by creating a *traditional folder*. The preferred way to structure the content is via *views*. A view can be thought of as a saved search. It has conditions for which objects to include in the view. The view updates automatically to include only objects matching the search conditions of the view. [39]

All objects belong to a single repository in M-Files. A repository created in M-Files is called a vault and is managed by the M-Files Server application. M-Files can connect to external repositories to display and manage objects in them with some limitations. The objects in an external repository may be unmanaged, meaning that they have no metadata available in M-Files. Unmanaged objects can be promoted to managed objects in M-Files by adding metadata to them. Examples of external repositories include SharePoint, Salesforce and network folders. [39]

The architecture of M-Files is a client-server model. Multiple clients connect to one server,

and the server contains all the information stored in M-Files. The server stores the information to its database and manages third party information in external repositories. The server can be on-premises, a cloud-based subscription service or a hybrid solution of both. [39]

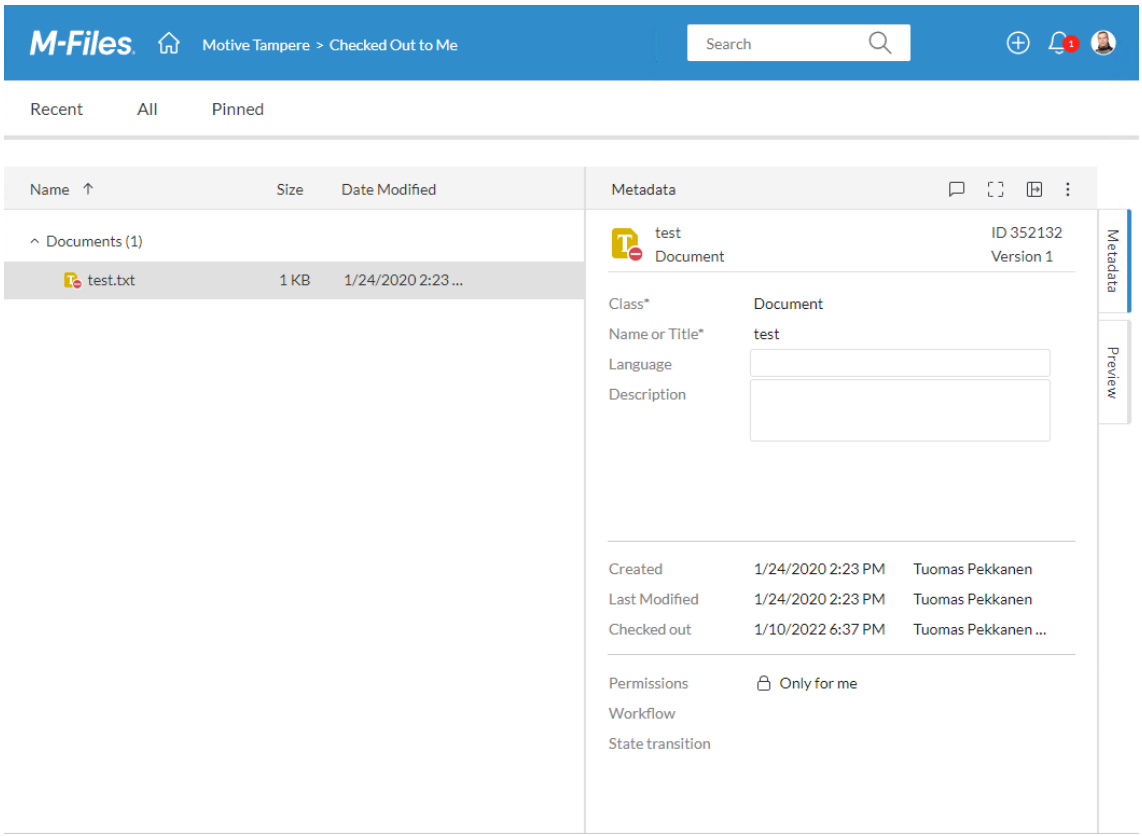

Figure 3.2. *M-Files web client in a browser.*

In figure 3.2 an example of the M-Files web client is shown. In the example the client is not in an embedded environment, it is accessed directly from the site hosted by the M-Files server. As the client has the full page of the site to itself, it shows all the UI elements in their expanded form. The web client is developed with modern web technologies and frameworks, but can be accessed from browsers not natively capable of using these. To enable access for older browsers as well, the client code is transpiled to ECMAScript 5 and polyfilled [39].

3.2 Current Embedding Methods

The previously mentioned connector approach refers to M-Files accessing objects in external locations. The integration is also possible the other way around, which is the focus of this thesis. Other applications can embed the M-Files web client, which enables accessing the M-Files user interface directly from the external application. The external application usually contains a page, tab or some other container which hosts the M-Files

web client in it. When the client is in the context of an add-in it changes to an embedded mode where certain UI elements are modified to fit in smaller spaces, or modified to fit the host content more appropriately. Examples of applications that have M-Files web integration available are SharePoint, Salesforce and Microsoft Teams. [39]

In figure 3.3 the M-Files web client is shown within Microsoft Teams. The web client is embedded with the *M-Files Add-in for Teams and SharePoint Online*. Compared to the full web client the web client is in embedded mode. Changes can be seen in the top bar styling and sizing.

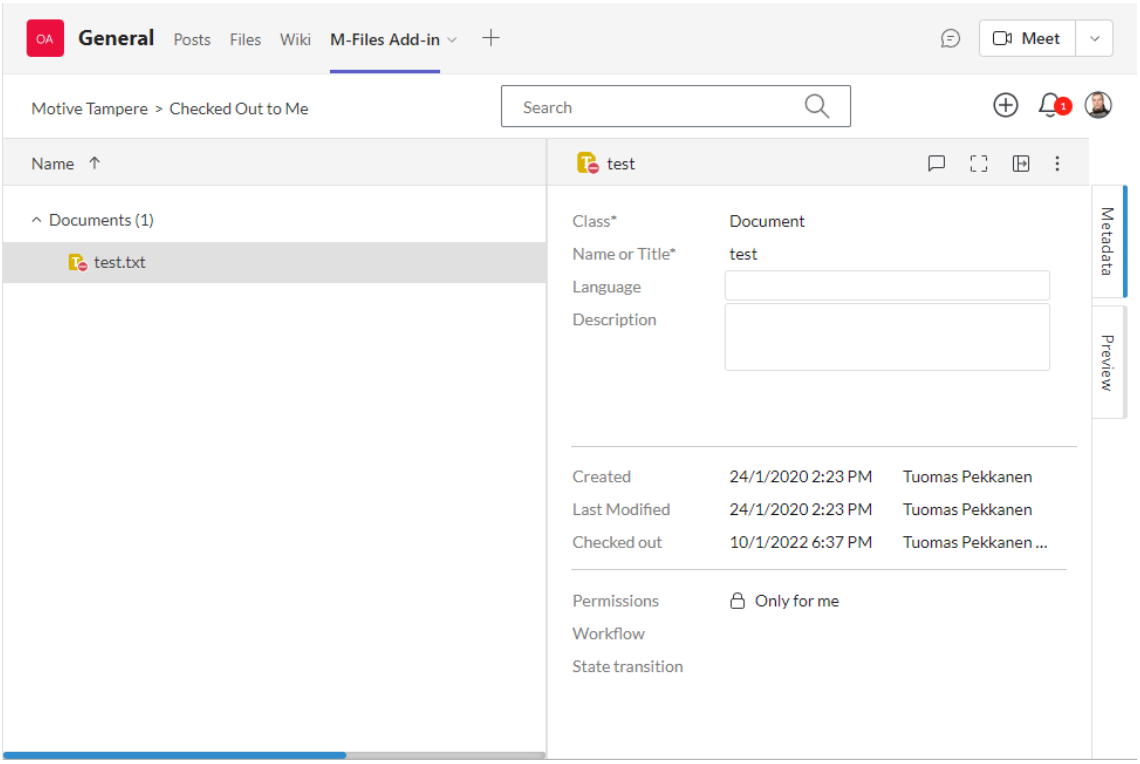

Figure 3.3. *M-Files web client with the Microsoft Teams add-in.*

The integration to an external service in practice means uploading an application bundle provided by M-Files that contains the web client with platform-specific integrations. Embedding the bundle does not require any code modifications as adding the application bundle is usually done through a user interface of the platform. To configure the settings M-Files needs to function correctly, the application can be modified with the configuration interface provided by the platform [41]. The interface is different for each platform, but the settings they send to the M-Files application are mostly the same.

In the Teams add-in shown in figure 3.3 the panel can be opened from the settings button of the M-Files add-in. The button is accessible through the arrow icon on the top next to the add-in name and opens the configuration panel to the right of the application.

The panel has settings for *server URI*, *vault GUID*, *view URI*, *keyword* and *editing mode*.

The server URI, or server uniform resource identifier specifies the M-Files Server the web application connects to and shows data from [41]. The second configuration parameter is the vault GUID [41]. The vault GUID uniquely specifies the M-Files vault the web client connects to in the server [41].

The third and fourth parameters are optional, and only one of them can be in effect at a time. The view URI can be specified to have the web client open to the chosen view when opening the M-Files client. The URI contains the path to the view and is composed of view IDs. The fourth setting is a keyword. If set, the M-Files client opens to a search of content containing the keyword when opened. [41]

The final configuration option is the editing field. It is used to control whether the embedded client should be opened in read-only mode, where no new objects can be created and existing objects cannot be edited [41]. The editing field setting is in addition to other M-Files permission configurations, and does not grant new permissions to users who do not have the permissions to edit the objects already [39].

The external services embed M-Files using platform exclusive integration methods. The embedding is done by M-Files and requires additional work for each new integration made. The web client needs to be integrated to use features offered by the platform and bundled into an application package only embeddable on that specific platform. The platform-specific integrations enable a deeper connection to the platform which leads to some advantages. But as each platform requires development time from M-Files, it is not possible to support every platform this way. Customers of M-Files may still want to embed the M-Files web client to their own product. For this purpose it is desirable to enable the external parties to embed the M-Files web client in a more generic manner. Such generic approach naturally cannot benefit from various platforms' specific integration features, but should enable third parties to embed and configure the web client enough for normal usage of M-Files.

3.3 Objectives for Embedding

The objective of this thesis is to create a generic way for external parties to embed the M-Files web client effortlessly. The method should be generic in the sense that it can be embedded to practically any host page environment regardless of technologies the site is using or other content it has. The purpose of this generic embedding method is to enable embedding M-Files to sites that do not have a dedicated add-in implemented for them.

Technically third parties could directly embed the web client using an iframe, but the end result would not be practical. If the application is embedded using an iframe with no additional tuning, it would behave as if it was hosted on a full web page. The web client does not know that it should be run in embedded mode, as it does when embedded with

the application bundle produced for a supported external platform like Salesforce. This leads to several issues with using the web client.

The first issue is the user interface of the embedded web application. The web client shows an extensive user interface with multiple sections, which are designed for use in an environment with lots of space to show elements in. If the web client designed for a full-screen experience is put into an iframe in a smaller portion of the hosting page, the space available would not be used properly. The content shown would include a relatively large top bar with a logo, a search bar and buttons for other functions. In a confined environment the search bar and button elements might not show properly since the logo is drawn first, taking most of the horizontal space available. Another issue in small screen spaces is the amount of content shown. There are multiple sections to the web client such as current search results, a metadata view and a preview of the current document. Trying to show these in small spaces would lead to only seeing one of them at a time, or all three at once if the elements are scaled to an uncomfortably small size. The same issues apply to environments where the screen is smaller even if the application is embedded to take the full space of the page.

The second issue is the lack of easily accessible configuration. Since the existing configuration panels of M-Files web client are tailored for each external platform, they are not available for the embedded M-Files web client in unsupported platforms. Without the configuration panel the most commonly used and mandatory options of the M-Files web client cannot be easily set. These mandatory options include the server URI, vault GUID and editing mode. The optional parameters are the view URI and the keywords. All of these parameters should be effortlessly accessible to the embedding party without needing deep technical knowledge of the embedding process.

Both of the issues mentioned could be solved by a third party with web development experience but would require significant implementation effort. The user interface running in the iframe could be modified by manually removing and resizing elements with scripts. The complexity of the modifications depend on the constraints from where the client is embedded to, but can not be considered an effortless task regardless of the complexity. The missing configuration panel options could be programmatically transferred to the web client without the panel user interface, but changing the options later is laborious compared to a user interface where the options can be directly input. Users without technical background would likely not be able to edit or configure the settings themselves. Implementing the communication requires knowledge about communicating with an iframe, and detailed understanding of the configuration format passed to the web client. Without these the application would embed in the full mode instead of the embedded mode.

Solving these issues needs a new way of embedding the M-Files web client. The web client should be effortlessly embeddable by third parties wanting to extend the usage

platforms of M-Files. Being effortless to embed in the case of this thesis refers to the complexity of adding the embeddable web client to the third party's own context. The embedding should only take a few lines of code and the configuration needed from the third party kept to minimal. The most commonly used settings should be accessible without needing to change any code. Being effortless also requires the embedding solution to work in multiple different contexts. The third party may want to embed the application in practically anything capable of running JavaScript. To be effortless to use in different environments, the embedded application should support the vast majority of the possible environments.

4. POTENTIAL FRAMEWORKS

This chapter presents potential frameworks for use in the implementation phase. Each framework is presented and then compared against each other. The comparison is done using common criteria for all the frameworks that is set in the beginning of this chapter. One framework is selected to be used in the implementation phase of the thesis.

4.1 Framework Selection Criteria

To select a framework used in implementation, common criteria to compare the frameworks are set. A necessary criterion for all frameworks to be included in the comparison is the ability to produce a JavaScript-based library, element, component or bundle which can be embedded to third-party websites. Frameworks without this capability have been excluded from the comparisons as they cannot produce an embeddable result. For simplicity the produced library, element, component or bundle will be referred to as *application* from this point onwards.

The first criterion is the ease of adding the application to a third-party website. In other words, the application should be effortless to embed by the third parties. The second point of comparison is the longevity of the framework. The framework should have developer support now and in the foreseeable future. The support is considered for example from the age of the framework, the developers and their amount and the possible company behind the framework. The third requirement for the frameworks is their browser compatibility. The frameworks should work on at least the following major modern browsers: Edge, Safari, Chrome and Firefox. The browser support depends heavily on the ECMAScript standard support and the availability of polyfills for older platforms. Old or legacy browsers in this thesis refer to browsers supporting at least ECMAScript 5. Internet Explorer 11 is used only as an example and the support of it is not a strict requirement. Browsers released before Internet Explorer 11 are not included in old or legacy browsers in this thesis. The final measure is the flexibility of the framework. The framework is considered flexible if it is independent of other frameworks or tools and can be extended with other frameworks or tools without compatibility issues. This ensures that the framework is able to react to changes in the other frameworks or tools used and does not force the usage of some set of frameworks or tools that would otherwise not be required.

As there are multiple frameworks available for comparison a prescreening is made for each framework. The purpose of the prescreening is to select frameworks that have the most potential for a more detailed comparison. Passing the prescreening requires that the framework does not have any major issues that can be immediately deemed to be a reason to not choose the framework. The reason can be from the criteria mentioned or from outside them. The frameworks that pass the prescreening have a simple test project made on them to see the practical usage of the framework and to find potential new issues. The frameworks that passed the prescreening and have the test project made with are compared to the criteria set, and one framework is chosen to implement the embedding project with.

4.2 Lit

Lit is an open source lightweight library for building web components. The components Lit builds are standard web components making them natively compatible with modern browsers. The produced component does not depend on tools or frameworks other than ECMAScript 2019. Because the component produced by Lit does not have dependencies to other tools or frameworks, the component can be shared to third parties regardless of the environment or frameworks they are using [42]. From the third-party point of view embedding a Lit component requires two lines of code: A script element with reference to the produced component and the new element produced by the component [42]. The script element loads the produced JavaScript which creates a definition of a new element to use in HTML.

If Lit is only used by providing a npm package and used in modern browsers it does not require any configuration, assuming JavaScript is used [42]. Lit itself does not provide any bundling, transpilation or polyfills but can be extended with other tools to do so. In addition to JavaScript Lit supports TypeScript [42]. TypeScript can be added to the project, but the developer has to configure the transpilation to JavaScript as Lit does not include TypeScript in it. The output of Lit can be configured using bundlers such as Webpack or Rollup [42]. The bundlers compile multiple code files to one single file which can be easier to share than multiple small files. With bundled code the third party only needs a single script file to include that has all the files combined into it. Without a bundler multiple files may need to be included by the third party. The bundling phase can be customized to minimize the code, transpile it to earlier ECMAScript versions, add polyfills to it for legacy browser support and as a last example to inject a service worker into the code [42]. A service worker is a proxy between the web application, browser and the network which can enable offline functionalities to the application by caching the state to the browser [43]. Lit does not restrict the tools used. The bundling can be done with any tool necessary as well as other customizations. It is also possible to not use a bundler

and manually handle the other tools, such as Babel for transpiling [42].

```
1 <body>
2   <hello-world msg="Hello"></hello-world>
3   <script type="module" src="<URI>/hello-world.js"></script>
4 </body>
```

Program 4.1. *Importing a component generated with Lit in a modern browser.*

Browser support of Lit depends on the possible external configuration. With modern browsers capable of running ECMAScript 2019 and web components as a target, only bare modules need to be configured. Bare modules refer to package imports in JavaScript using the npm package name as the source instead of a URL or a path to the package. Webpack and Rollup are able to transform these for browsers. The modern browser support of Lit is shown in table 4.1

Table 4.1. *Lit ECMAScript compatibility with modern browsers [42].*

Browser	Supports ES2019 & web components
Chrome	>=73
Safari	>=12.1
Firefox	>=63
Edge	>=79

To support legacy browsers as defined by Lit the code needs to be transpiled to ECMAScript 5, the ECMAScript module imports converted to another module system that is supported by legacy browsers and the needed polyfills to be loaded. The polyfills recommended for legacy browsers include the web component polyfills with additional Lit polyfill layer, standard JavaScript library polyfills and Promise polyfills. The legacy browser support stated by Lit is shown in table 4.2. [42]

Table 4.2. *Lit ECMAScript compatibility with legacy browsers [42].*

Browser	Transpiled	Transpiled & polyfilled
Chrome	67-79	<67
Safari	10-12	<10
Firefox	63-71	<63
Edge	79	-
Edge "classic"	-	<=18
Internet Explorer	-	11

The library was first released in 2017 and has since seen frequent releases [44]. The library is maintained by Google employees and the community of open source contributors [44]. By NPM package manager downloads the currently recommended *lit* package has around 200 000 weekly downloads, with the legacy *lit-element* and *lit-html* packages having around 750 000 and 850 000 weekly downloads respectively [45]. The documentation Lit provides and the community support is the most comprehensive of the frameworks tested.

Lit was selected to continue from the prescreening phase. No issues were found during the initial research that would suggest Lit not being a suitable framework to implement the thesis project with.

A test project was made with Lit to verify the functionalities. The project contained a simple web component with text elements and one parameter passed from the embedding site. With Chrome, Firefox and Edge the result is fully functional. To test legacy browsers transpilation and polyfills were introduced to the project with Rollup. The project was targeted to ECMAScript 5 with polyfills for web components. The transpilation result with Babel 7 was compatible with legacy browsers, but the polyfills had configuration issues. The configuration issues should be solvable, but require more time than spent in the initial investigation.

4.3 Hybrids

Hybrids is a framework for creating web applications and web components. It is open source and uses the standard web components API. A difference to the other web components tools mentioned in this thesis is that Hybrids is based on plain objects and pure functions. The other web components mentioned are based on classes and not always pure functions. The framework does not restrict the frameworks or tools that can be used with it. [46]

While Hybrids is open source, almost all the contributions to the project are made by the same author. The first release of Hybrids was in 2016 and has been frequently maintained since then [47]. The framework is not widely used, having weekly downloads of the package at around 4000 [45]. Issues of Hybrids in the GitHub page are actively inputted and solved. Currently, there are no open issues [47]. The support documentation of Hybrids is extensive. The documentation includes a migration guide from different versions, has a well documented API and explains the basic technical principles behind Hybrids such as the component model [46].

The browser support of Hybrids is on par with the other frameworks for modern browsers. In the newest main release *v8.0.0* Hybrids has dropped support for web components polyfill [46]. Polyfill support is one of the main criteria for the frameworks in this thesis. In

addition to not having polyfill support the framework has relatively low usage compared to the other frameworks and depends mostly on the single maintainer. Hybrids was dropped during the prescreening for these reasons.

4.4 Snuggsi

Snuggsi is an open source web components framework that specializes in being easy to develop with [48]. Snuggsi does not require any installation, instead it recommends using a script including element in the site embedding the web component [48]. The generated web component is based on the standard web component API, making it compatible with other frameworks and tools [48]. Developing with Snuggsi does not require other tools since it transpiles and polyfills the web component as needed by the browser accessing it [48]. As Snuggsi includes the tools itself, the configurability of it is limited. The tools used internally cannot be accessed without modifying the source code of Snuggsi itself, which makes possible future configurations harder to implement.

The browser support of Snuggsi states covers only the most recent versions of browsers, except for the stated Safari support from version 9 onwards [48]. With transpilation and polyfills the support should also cover older versions of the browsers, but this support is not mentioned anywhere. The documentation of Snuggsi is limited. The documentation provides a few usage examples and usage tips that all fit to the project repository's introduction [48]. The official website has pages missing content and the API is introduced only by function names commonly [49]. The contributions to Snuggsi code base are mostly made by the personal author of the framework itself [48]. There are recent updates to Snuggsi, and it seems to still be maintained. The GitHub page activity related to issues and new features have slowed down during the last year [48].

To include a component generated by Snuggsi more effort needs to be done on the embedding site [48]. Using the component is similar to what Lit does, but including the source code has differences. The examples include the Snuggsi package itself in the HTML which adds another line of code. In addition, the whole component code is included in HTML [48]. It should be possible to bundle the component using a bundling tool such as Webpack or Rollup, so that the component code itself does not need to be copied.

Snuggsi did not pass the prescreening. The framework is maintained but the future of it is unclear. Activity in the framework's development shows signs of decelerating. The framework is mostly dependent on a single maintainer. The documentation is limited, and the official website is not professional with content missing. Embedding the content generated with Snuggsi is not as effortless as with the other frameworks. The uncertainty of the framework's future and missing documentation are the main reasons for not proceeding with Snuggsi.

4.5 Stencil

Stencil is a toolchain for building web components. It is an open source set of tools that compiles TypeScript to a standard compatible web component. Using JavaScript directly is not supported. The generated component is compatible with other frameworks and tools. Embedding a component generated with Stencil is identical to Lit with the example code in figure 4.1. [50]

Compared to the previous frameworks, Stencil is a compiler. Internally the toolchain adds polyfills, transpiles TypeScript to JavaScript, minifies and bundles the code. The result is ECMAScript 2017 based for modern browsers and ECMAScript 5 based if configured to support legacy browsers. Stencil uses Rollup internally, but allows Rollup plugins to be added from configuration files as opposed to Snuggsi which does provide a way to modify the plugins used by its internal tools. While the internal tools cannot be completely changed, they can be relatively easily expanded by adding plugins to them.

Stencil is developed by Ionic and the open source contributors [50]. It has frequent releases, with the oldest currently available release being from 2019 [51]. The Stencil npm package has around 330 000 weekly downloads [45]. The documentation of Stencil is comprehensive, listing support policy of the Stencil releases, API documentation and examples [50].

Browser support of Stencil from the documentation is seen in table 4.3. Stencil does not mention the transpilation requirement in the document. It only states the component having “full native support” [50]. As the components are written in TypeScript they must be at least be transpiled to JavaScript before browsers can execute the programs. Stencil loads polyfills for legacy browsers dynamically as needed, so that only the functionalities missing from the browser are polyfilled [50].

Table 4.3. *Stencil ECMAScript compatibility with browsers [50].*

Browser	Transpiled	Transpiled & polyfilled
Chrome	>=60	-
Safari	>=10.1	-
Firefox	>=63	-
Edge	>=79	-
Edge "classic"	-	16-18
Internet Explorer	-	11

Stencil passed the prescreening. The only potential issue found with Stencil is the flexibility of the tool in respect to the internal tools it uses. This issue is not major enough to not

pass the prescreening, since the internal tools by default do have transpiling and polyfill support. In addition, the tools can be configured by modifying their plugins.

A test project was implemented with Stencil. Compared to the interpreted solutions the compilation phase adds a few seconds of delay between changes and the result updating to browser. The example project generated by the Stencil starter package includes examples of the component code, unit- and end-to-end tests, automatically generated documentation and configuration files. The component created was similar in functionality to the previous test projects. In modern browsers the component functioned according to expectations, as with the previous projects. To support legacy browsers the Stencil configuration was modified to target ECMAScript 5 with the following polyfills: *cssVarsShim*, *dynamicImportShim*, *shadowDomShim*, *safari10*, *scriptDataOpts*, *appendChildSlotFix*, *cloneNodeFix* and *slotChildNodesFix*. With the modifications legacy browsers have identical behavior to modern browsers in the example project.

4.6 Selection of the Framework

The prescreening phase was passed by 2 frameworks: Lit and Stencil. A test project was made with each framework to verify their suitability to the actual implementation project. The frameworks are compared to the criteria set before the prescreening phase:

- Effort required for embedding the result
- Longevity and support of the framework
- Browser compatibility
- Flexibility

The embedding effort the third party needs to do is identical for both Lit and Stencil. Both frameworks are built on the web components standard and have identical methods available for embedding the content created with them. While the new HTML element usage is the same for frameworks that use the web components API, the way the framework itself and the component created with it are included can be very different. For Lit and Stencil the outcome is identical from the embedder perspective, 2 or 3 lines of code depending on whether legacy browser support is included. The embedder is not required to install or add anything else to their site, making the embedding effort for the base component minimal on both frameworks.

The longevity and support of the frameworks assess the confidentiality that the framework will be supported currently and in the future. Both Lit and Stencil are open source. Lit is developed by Google and Stencil by Ionic. Having a company behind the framework establishes more trust that there will be maintainers for the framework in the future. Both frameworks have frequent updates to them and neither company has shown signs of the

framework retiring in the foreseeable future. In the GitHub pages of the frameworks there are frequent new issue listings and solved issues. Lit has a larger user base according to the npm package downloads. The download counts of both platforms have been growing in the past year. The documentation of the frameworks are both comprehensive enough that it does not differentiate the two frameworks from each other. Overall, the longevity of Lit is considered slightly better because of the higher user base and Google being a significantly larger company backing the framework. It should be noted that Stencil is not an unsuitable framework, it also has a high probability of being developed in the future.

Browser compatibility of Lit is shown in tables 4.1 and 4.2. The compatibility of Stencil is shown in table 4.3. The compatibility of Lit depends on the configuration. It can be directly used without transpilation or polyfills to support only modern browsers. If polyfilled and transpiled the browser support is expanded to the one seen in table 4.2. In the case of Stencil the only option is to always at least transpile the code from TypeScript to JavaScript. If polyfills are added the support is expanded as seen in table 4.3. Both frameworks support the modern browsers similarly. For legacy browsers Lit does not list a minimum version that is supported, while Stencil promises compatibility starting from lower versions. The minimum of Lit is dependent on the tooling outside of Lit used to transpile and polyfill the result, but also on the polyfills provided by Lit to polyfill Lit itself. Both frameworks offer great browser compatibility for modern and legacy browsers. Of the browsers provided Lit has equal or better support for Safari, Firefox and Edge. Stencil has possibly better support for Chrome and pre-chromium based Edge "classic". Since the advantage of Stencil for these browsers is uncertain and Lit has 3 browsers out of the 5 better supported, Lit has the better suitability for the implementation project from browser support perspective.

The final criterion for the frameworks is their flexibility with other tools. Lit and Stencil are agnostic to other UI frameworks. The components they produce are compatible with sites made with React, Angular and Vue as some examples. Where the frameworks differ is their internal tooling. Compared to Stencil, Lit is a plainer framework. Lit itself does not include transpiling or polyfills. Stencil is more of a compiler that does transpiling and polyfilling itself. This makes Stencil easier to get started with, but Lit more configurable. With Lit it is easier to add or modify the other tooling used. The same applies to the bundling of the application. Stencil does this itself with Rollup requiring no configuration from the developer. Lit supports Rollup, but also other bundling tools such as Webpack. With Stencil the language is forced to be TypeScript, while with Lit either JavaScript or TypeScript can be used. The flexibility of the framework is an advantage to Lit. For the purposes of this thesis Lit is more suitable with the freedom to configure it as needed. If the project needs a framework that requires less configuration and is more focused on getting started quickly Stencil can be the better choice.

The framework chosen for the implementation is Lit. Lit has the advantage in longevity,

browser compatibility and most prominently the flexibility with other tools. The flexibility of Lit is preferable for the thesis project as Lit allows using JavaScript which Stencil did not, and can be modified to use other tools in the future as needed. With Stencil there is limited configurability by using plugins on the internal tools, but the internal tools cannot be completely changed. If for example the bundling solution needs to be changed, that is only possible with Lit.

5. IMPLEMENTATION

In this chapter a solution for embedding the M-Files web client is made. The web client is included in a web component produced with Lit. The web component will have a way implemented for it to communicate with the M-Files web client. The produced component is transpiled, polyfilled and bundled to a JavaScript file that can be used by the third parties to embed the M-Files web client effortlessly.

The chapter explains the most important parts of the complete program A.1 in the appendix. The program contains more implementation details than explained in the example parts. The examples are simplified to showcase only the necessary features.

5.1 Structure of the Component

Making a solution that is effortless to embed, but also secure requires planning from the architectural level. If the web client is embedded with a script directly, it would be easy to modify but would also have issues. Content embedded with a script is not completely isolated even with DOM encapsulation from the Lit framework. The web client still uses the same browser storage for example, which leads to conflicts if multiple web clients are wanted on the same page. On the other hand, if the content is embedded in an iframe the configurability is more limited, since the iframe only allows controlled and predefined messages through. The only method of changing the configuration would be with code changes, which is not practical. With script-based embedding it is possible to create a configuration panel similar to the ones already existing with the SharePoint, Salesforce and Teams add-ins.

To gain both isolation of the complex application and to have an easy way to configure it, a hybrid approach is used. The web component itself created with Lit is embedded with the script-based embedding. Parameters can be passed to the web component easily without restrictions from an iframe. The component can be extended to offer an API for more control if necessary. It is possible to create a configuration panel outside the web client that can be used by end users to pass configuration to the web client. With the benefits of script-based embedding, the web client itself is embedded with an iframe inside the component. The third party embeds the component using script-based embedding, and the component contains an iframe as one element of it. This approach

ensures that the web client inside the component is in an isolated environment inside the iframe, but the component itself is effortless to configure. The component handles passing the configuration to the iframe in a controlled manner, creating an abstraction layer to the third party.

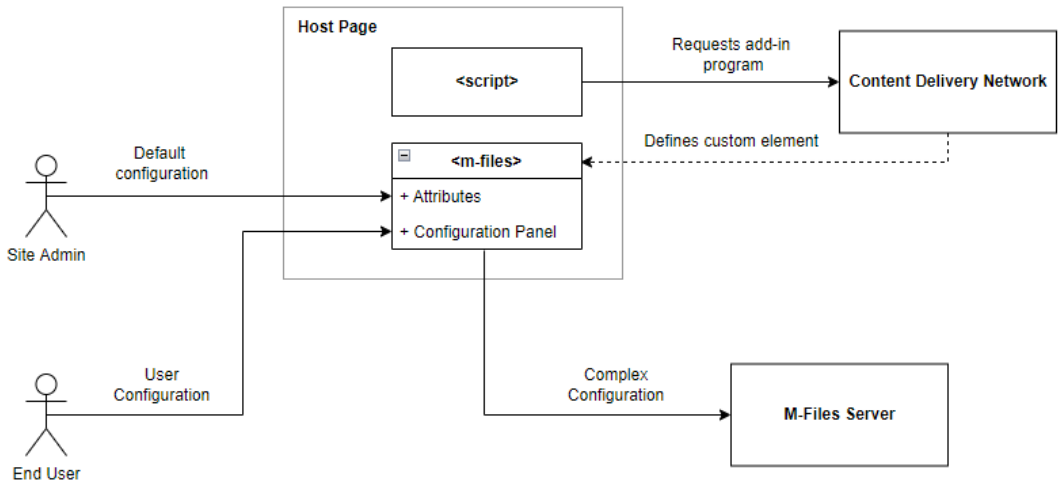

Figure 5.1. *Overview of using the generic add-in.*

Figure 5.1 shows an overview of how the add-in behaves from a third party perspective. The add-in is composed of the script and m-files elements. The script contacts a content delivery network to obtain the web component script. The script defines the custom element and its functionalities.

The m-files element abstracts the complicated structure. The element contains the iframe, configuration panel and a method for them to communicate. As the element handles these the only things exposed to the third parties are the attributes of the element and the configuration panel rendered in the browser. Admins of the third party can configure the element using attributes to give defaults for all end users. The end users can make their own modifications from the configuration panel to have user specific customizability. The element uses these simple exposed configurations to create the complex configuration required to embed the client. The complex configuration includes communicating with the iframe and correctly initializing the client.

Lit renders the component from HTML in a *render* function [42]. The M-Files web component's render definition can be seen in program A.1. The HTML passed is in the form of a *template*. The template can include dynamic content with *expressions*. These expressions are marked with a $ sign and can include reactive properties. The component is split into 2 parts, the *contentTemplate* and *configurationPanelTemplate*. The *contentTemplate*

contains the M-Files web client and a loading screen. The *configurationPanelTemplate* has a configuration panel which allows users to modify the configuration from a graphical user interface.

Reactive properties are Lit's method of managing the component state. The properties can be public or private. Public properties are exposed to the outside via the element attributes by default. These properties are can be used by the site adding a component created with Lit to pass parameters. Private properties are for internal use of the component and cannot be modified from the element attributes. Reactive properties are used in Lit to define rules for rendering the component. The properties can be included in expressions giving attributes to the component's inner HTML elements. They can also be used to selectively render selected parts of the defined elements. When a reactive property changes Lit re-renders the necessary parts of the page to reflect the change. [42]

The component is encapsulated by Lit. The component is rendered to a shadow root. The HTML elements and styles defined with css are scoped to the shadow DOM. The shadow DOM is an isolated and encapsulated inner DOM inside the hosting site's DOM. Querying elements in the host DOM do not return results from the shadow DOM and vice versa. Styles applied in the shadow DOM do not pollute the outside of the shadow DOM. This isolation and encapsulation enable using elements and styles without needing to worry about conflicts with the host page. [42]

5.1.1 Web Client

The web client is in the *contentTemplate* inside an iframe. The template is structured in a way that has the iframe containing the web client always rendered, but not necessarily visible. The iframe needs a source attribute to load the web client from the correct destination, but the source is not known at startup. As other pieces of the program modify the iframe source, it must exist in the DOM, but is hidden until it is ready to show the correct source. Hiding the iframe is controlled by the *_loading* private reactive property.

To prevent an empty screen or an error from invalid iframe source showing, a loading indicator is included in the *contentTemplate*. The loading indicator is controlled by the same *_loading* property, but the indicator is completely removed from the DOM when the iframe is ready to show the M-Files web client. Using the reactive property removes the need to manipulate the DOM by the developer since Lit does the necessary updates to the shadow DOM in batches for optimal performance.

5.1.2 Configuration Panel

The *configurationPanelTemplate* contains HTML definition of the user interface for configuring the web client. For consistency the configuration panel has a similar look to the

customized add-ins but is themed to fit the web client. The customized add-ins have implemented a way to open the configuration panel outside of the application. As the generic implementation created does not have such context to open the panel from, it is opened from the component itself. A settings button is added floating on top of the iframe, positioned next to the notification icon in the top right corner of figure 5.2. The button is used the open the configuration panel after the initial configuration.

The configuration panel has inputs for configuring the web client, exactly as the customized add-ins have: server URI, vault GUID, view URI, keyword and the editing mode. The values are stored to the properties of the component immediately when a change is detected. These mappings to properties are managed by the event handlers given to each input element. These properties are not instantly transferred to the web client. Instead, similar to other add-ins the configuration panel has a confirmation button *apply* that is responsible for passing the parameters forward. The apply button has an event handler that applies the configuration of the iframe source. The iframe is modified in the shadow DOM to change the *src* to the initialization page of the web client corresponding to the provided server URI.

The iframe source would in the best scenario be controlled by a reactive property, but the reactive property approach has a major limitation here. If configuration is changed in any combination except for the server URI remaining the same, the reactive property would not change and Lit would not request an update to the component. The only time frame when the web client accepts a configuration is during the initial loading which would not trigger without the iframe source updating. Solving this flaw requires modifying the source attribute manually without a reactive property, always forcing an update to the iframe. Compared to using a reactive property, manually changing the source is more complicated and requires attention to when the iframe is actually rendered in the component. The solution is acceptable, but not optimal from maintainability perspective.

5.2 Communicating with the Web Client

As the web client is isolated in an iframe, the communication with it should be done with the postMessage API. Using postMessage the iframe content can safely accept input from sources it trusts and ignores other communications. When the iframe is in the initialization page it accepts communication with postMessage from the web component. This communication is implemented in the *__clientReadyHandler* event handler registered to the iframe. The event triggers when the iframe has loaded the content from the source. The event is dispatched even with an empty source when the iframe is first loaded. To prevent sending messages to an empty source, the event handler has checks that terminate the function early if needed.

When the iframe has loaded the initialization page it first disables the loading indicator.

As the disabling is handled using the reactive property *_loading*, Lit modifies the DOM automatically. If not using a framework with dynamic rendering capabilities, the DOM modifications are the responsibility of the developer and would be needed in this function. Disabling the loading indicator reveals the web client running in the iframe, the visibility of which is also controlled by the *_loading* property.

At this phase the web client is waiting for a message from the component. The component prepares the configuration that will be sent to the web client in the iframe. The configuration is presented as pseudocode in the variable *config*. The configuration includes parameters from the configuration panel, which are stored in properties of the component. The configuration is then passed to the web client using *postMessage*. The first parameter of the *postMessage* is the configuration in pseudocode. The second parameter is the target origin, which restricts the allowed receiver of the message as a security measure. The message can only be received by the origin defined as the server URI in the configuration, otherwise the message is not sent. This prevents the message from being sent to a potentially malicious site, if for example the user has navigated to another site in the iframe. The current origin of the iframe must match the target origin specified in the *postMessage*.

When the web client receives the message it uses the configuration when preparing the main M-Files web application. The client is configured to run in an embedded mode, removing elements from the UI to fit in a smaller space. The client accepts the parameters from the configuration. It opens the vault corresponding to the vault GUID from the server specified in the server URI. The optional parameters view URI and keyword control the default search or view that is shown when the application opens. The editing mode configures the client whether editing options for objects and views are shown. The parameters can be changed at any time, triggering a new initialization phase of the application to accept the changed parameters again.

5.3 Cross-Origin Resource Sharing

The host page containing the web component and the server hosting the M-Files web client are usually from different origins. Communicating with different origins is restricted by the M-Files server. If the iframe tries to load the web application from the server, the connection is by default rejected as part of the cross-origin resource sharing policy. This policy prevents all but explicitly allowed origins from showing the site. Disallowing unknown origins is beneficial when preventing unofficial channels from consuming network traffic. Allowing all origins is also a potential security threat. If the unknown origin is malicious it could implement a clickjacking attack where the user is tricked into sending the credentials to the malicious party from an invisible button on top of the actual login button. To prevent clickjacking a content security policy is set by the server which prevents the

web client from being embedded in an iframe from different origins, except ones that are whitelisted.

Without access to the M-Files server hosting the web client, these settings cannot be modified. Only the administrators in M-Files server have access to these settings. To use the web component the advanced vault settings in M-Files Admin have to be modified. Under M-Files Add-In Settings is an option for configuring the allowed cross-frame sites. This setting is modified to include the origin of the page wanting to host the web component. This setting manages the cross-origin access and the content security policy.

In the prototype implementation the M-Files web client is hosted at *http://localhost:7767*. The site embedding the web component is hosted at *http://localhost:8000*. To enable using the web component, the M-Files Admin is configured to allow connections from *http://localhost:8000*. With this configuration the web client can be accessed directly in its full form from *http://localhost:7767* and in embedded form from *http://localhost:8000*. All other origins trying to embed the web component or the application directly are rejected.

5.4 Preserving the Configuration

Once configured from the configuration panel, the application should retain the settings the next time it opens up. If the settings are found during the second time the web component is loaded, it can directly pass the parameters to the iframe. This improves the usability as the user does not have to type the configuration each time they reopen their browser.

To store the configuration the local storage of the browser is used. The local storage persists between browser sessions and contains key-value pairs of string-formatted data. The configuration panel's apply button has a function call to *storeConfiguration* which saves the configuration parameters defined in the configuration panel to the browser's local storage.

To load the configuration the *loadConfiguration* function is used. The function loads the parameters from the local storage and sets them to the current properties. This function is called when the web component element is connected to the document, overwriting the default parameters if the corresponding values are found in the local storage. As Lit manages these reactive properties the UI is automatically updated to reflect the changed values from the defaults. The loading is called from the *connectedCallback* lifecycle method which occurs after the constructor. If the *loadConfiguration* is called from the constructor, the values written would be overridden by Lit when it applies the attributes passed to the element. Lit applies the attributes only after the constructor is called. To make the user specific settings from the configuration panel have a priority over the element attributes from the page HTML, they load when the component connected to the document.

5.5 Build Tools

Sharing the application for use requires build tools to optimize the deployment. The build phase is required for ensuring compatibility with a wide browser base by transpiling and polyfilling the code. The build tools can also optimize the code by minifying it for a smaller download size. As the most important task to make the component easily shareable, the build tools bundle the files to a single file containing all dependencies and assets needed. The single file can then be hosted by M-Files or a content delivery provider and be downloaded from there by the third parties. A single bundled file is efficient to download, and easy to use with a single script file include.

Lit allows practically any build tool to be used with it. For the prototype implementation Rollup is used. Rollup is configured with an input file and an output file. The input file is the starting point of the application. The output is the location of a new file where the bundled result is placed. Including other features such as transpiling requires the use of Rollup's plugins. The plugins are executed in order after the input has been processed. After the plugins are done, the output is bundled.

5.5.1 Babel

The component is transpiled to ECMAScript 5, matching the compatibility with the web client. The transpiling is done with Babel using the settings preset *preset-env*. With no target option given, preset-env defaults to transpile to ECMAScript 5. The transpilation modifies the component code and the dependencies it imports from node modules to ECMAScript 5. The Babel settings are defined in *babel.config.json*. Babel version 7.18.0 is used.

Polyfills of the project are imported with Babel. The preset-env is configured to polyfill ECMAScript features from the core-js library. The *useBuiltIns* is set to *usage*, making Babel add the import statements to the polyfills in each file where they are needed. This can lead to duplicate imports, but the bundling phase of Rollup will remove the unneeded duplicates. Core-js version 3 is used for loading the polyfills.

Babel is not directly added to the project, it is instead used from the Rollup's Babel plugin. The plugin adds Babel support to Rollup and includes Babel in it. The plugin uses the Babel configuration file defined earlier. It is possible to set the Babel configuration in the plugin directly if needed.

5.5.2 Postprocessing

In addition to transpiling and polyfilling, Rollup is configured to process the code to a compact format. The processing includes 2 phases to minify the code. The first phase is

using the *minify-html-literals* plugin. The plugin minifies HTML and CSS inside template literal strings. Without this plugin the *html* and *css* Lit specifiers would not be minified as they are not recognized.

The second phase is using Terser to minify the remaining JavaScript code. Minifying the code changes the declared names in code to short terms and removes spacing of the code. It also removes comments and unnecessary semicolons leaving only what is absolutely required for the program to run correctly. This makes the code practically unreadable to humans, but also takes only a fraction of the space. Minified programs require less traffic to download which reduces the delay before the application is loaded.

5.6 Using the Component

After the bundling is complete, third parties can import the generated script file to their page and begin using the *m-files* element. As the bundled program has been transpiled and polyfilled, it can be included in a wide range of browsers. An example HTML body of a third-party embedding the web component is shown in program 5.1. In a production environment the bundled source file would be hosted on server and referenced from there instead of the local computer's memory. The content delivery network in figure 5.1 showcases this as it returns the bundled script.

```
1 <body>
2   <m-files editing="false"></m-files>
3   <script type="module" src="../m-files.js"></script>
4 </body>
```

Program 5.1. *Embedding M-Files web client with the generic add-in.*

The program has set a default value for the *editing* configuration option. When the add-in is loaded the attributes are used as the initial values of the configuration panel. Configuration options stored in the local storage when the user has applied the settings have priority over the attributes set in the element. This makes it possible for the company to have a set of default configuration values, but also leaves the end users the possibility of customizing the values to their needs. The attributes are optional and can be inserted partially or completely left out. If the attributes are not given, the program uses the default values set in its constructor. In figure 5.2 the client using the generic add-in is shown running in a browser. The site uses the HTML body shown in program 5.1.

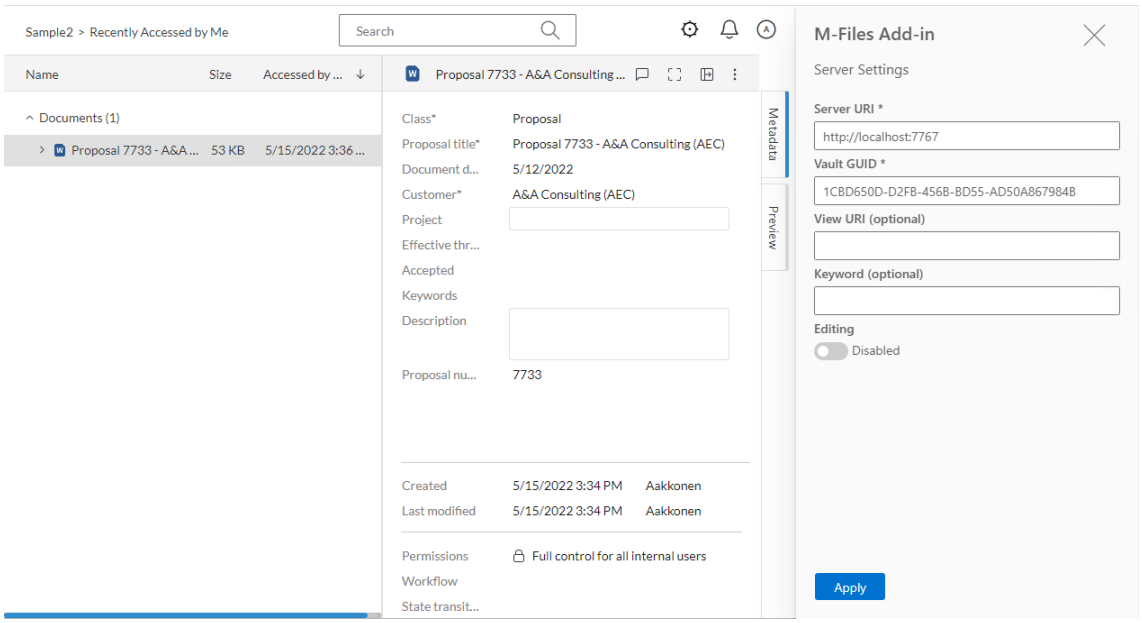

Figure 5.2. *M-Files web client with the implemented generic add-in.*

The page is a simple showcase containing only the created web component. In a third party environment the page may contain more elements as the component is added to an existing web page. The component can be added to a page regardless of other frameworks it uses because the component is not dependent on a specific framework existing in the host site.

6. DISCUSSION

This chapter explains the meaning of the results found in this thesis for effortlessly embedding complex web applications. The success of the implemented project is discussed. The thesis takes a look at potential future development for the implemented project and what could be gained from those developments.

6.1 Embedding M-Files

The thesis was set to find a solution to make it effortless for external parties to embed a complex web application. M-Files web client as a complex web application requires technical knowledge and significant effort to configure from third parties. The implemented solution makes this embedding process effortless to third parties. The implemented solution abstracts the technical knowledge and minifies the effort required to embed it by introducing a simple way of embedding and configuring the application. The proposed solution can be split into two parts. The first part of the solution is combining script-based embedding with an iframe. The second part is using a web component implemented with Lit as a container for the solution.

The first part combines script-based embedding with an iframe. The implemented script manages the iframe. The complex web client is isolated to the iframe environment and does not conflict with other applications on the site, from elements and styles to stored configuration parameters. As the program manages the iframe it removes the need for the embedder to know exactly how the M-Files web client behaves and what configuration it requires. Without the script managing the iframe the embedder would need complex logic seen in the program A.1 implemented by them. Without the solution the embedder is required to implement the complex communication with the iframe including loading the initialization page and correctly passing the configuration with *postMessage*. Such embedding process cannot be considered effortless.

The solution introduces an easy way to modify the configuration of web client. The prototype introduces a configuration panel which can be used by end users to change the settings without any technical knowledge of the embedding process that the program handles. The importance of this accessibility to configuration is crucial when making the component effortless to embed. Without such graphical configuration option inside the

running application the only way to modify the configuration would be by modifying the embedding code. Users of the application commonly do not have access to the website code nor the programming skills needed. With the configuration panel end users are able to configure the complex web application to their needs without the need to contact site administrators. Implementing a similar configuration interface can be considered to take such effort from the third parties that they would probably abandon it. The prototype implementation includes this configuration panel and requires practically no effort from the third party to take into use.

The second part of the proposed solution is using Lit as a framework and bundling the solution. These relate to how effortless from a programming view it is to embed the solution. Components created with Lit are standard web components that can be used in the same way as regular HTML elements. For the host site admins this means that there is no need to learn new frameworks or tools to embed the complex application. The solution can be embedded with 2 lines of HTML as presented in program 5.1. Changing the default configuration is optional, but effortless to do if needed by changing the element attributes. Bundling of the solution relates to the effortlessness in the *script* tag. The bundled solution script is a single file which can be loaded with a single line of code. Without bundling multiple script includes could be required, or more complex logic on the server serving the solution to return all the required files.

The thesis gives opinion on how suitable are the existing frameworks for implementing the solution. Both during the investigation of frameworks and in the implementation Lit is found to be a well suited framework. The investigation of different frameworks set criteria for them that can be used to compare the suitability of each framework. From this investigation both Lit and Stencil are considered frameworks that fill all the requirements. Choosing Lit instead of Stencil is based mainly on the need to have more control over the build tooling. Stencil is a suitable framework to use especially if the solution does not need or want to have fine-grain control of build tools, but would benefit from potentially not needing to maintain the build tools. Lit has better longevity and browser compatibility than Stencil, but Stencil is not considered unacceptable from those perspectives either. Implementing the solution with Stencil should also be possible, but Lit is preferred. The other frameworks researched in this thesis are found as not suitable for the solution. They each have at least one major issue that prevents creating a long-lasting solution that enables effortless embedding by third parties.

The implemented solution is a prototype. It is not meant to meet the strict requirements of production environment programs. The prototype demonstrates the feasibility of using the methods presented in this thesis to create a package that is easily embeddable. The prototype is considered a success and may see further development and productization in the future. Approaches to embedding complex web applications are found in the literature this thesis uses, but they use either the script-based embedding or an iframe [2]. The

solution proposed here is a combination of the two. Another solution detailed in the thesis is the use of a tailored add-in to specific providers using their embedding frameworks, such as SharePoint, Salesforce and Microsoft Teams [33, 36]. These have their own use cases with the program having access to the embedding platforms features. The solution of this thesis is not limited to a specific site or platform. It is a generic add-in configurable to a wide range of use cases.

6.2 Future Development

The prototype solution uses local storage for storing the settings configured by the end user. This approach is intentional and appropriate for the prototype, but will run into issues if multiple *m-files* elements are added on the same page. The local storage is not isolated to each element making the elements share the same local storage. If the user needs to have a user specific configuration for at least 2 elements at the same time, the configurations would overwrite each other when saved. This issue only applies to the element itself. The web client inside the iframe has its own local storage separate from the host page and does not have this issue. Solving the issue for just the element rather than the more complicated web application should be easier to do in the future.

There are plans to include a controlled communication channel for configuring the solution. Such channel would be added to the program A.1 to accept configurations from administrators of the company embedding the solution. This would make any code changes unnecessary for changing the configuration of the solution. At the same time it could eliminate the need for using local storage and the issue of embedding multiple web clients in the same page. The communication channel is out of the scope of this thesis and requires more detailed defining.

The prototype does not have host page context awareness on the same level the tailored add-ins have. Some context awareness could be introduced to the solution by exposing an API with which the third parties can communicate with the solution. The API could possibly be implemented with events from the element to the host page and vice versa. This could include sharing information from the element to the host site in which view the user is in now as an example. Similarly, the host page could request moving to a certain view in the element as a reaction to something occurring on the outside of the element. The examples given are only indicative and not a guarantee of what would be implemented.

The polyfills included in the prototype apply only to the ECMAScript features. The web components used by the Lit framework should be considered to be added in the polyfills. If web components polyfills are added the solution would work on browsers that do not have support for web components otherwise. These polyfills require more investigation as they might break the encapsulation provided by the web component to the styles and

elements hidden in the shadow DOM.

The frameworks compared are a small selection of frameworks available. Some framework not included in the investigation phase could be an even better candidate than Lit is. An interesting continuation study could be including more and vastly different frameworks for the comparison. Even with the limited selection of frameworks the thesis was able to find a framework that meets all the set criteria and is suitable for the implementation phase.

7. CONCLUSION

Complex web applications require significant effort to be embedded by third parties. Embedding M-Files web client to an external site is only possible with technically adapt personnel and tremendous implementation effort. In the case of M-Files the embedder needs to take into account initializing and configuring the application. The embedder is responsible for programmatically modifying the application's user interface to fit in the more confined host page environment. Without these modifications the M-Files web client is run in its full version. Compared to the embedded mode the full version takes more space and includes elements possibly not needed in an embedded environment. Modifying the application's configuration later requires code changes which hinders the usability. The thesis proposes a solution that is effortless to embed by third parties to external sites.

M-Files has created add-ins for SharePoint, Salesforce and Microsoft Teams. These add-ins can only be used on the platform they are made for and configure M-Files web client to those platforms specific needs. The solution created here is not meant to replace these, but to be in addition as a generic add-in that can be used in almost any website. As a generic solution the created program is not required to deeply integrate with the hosting website's special features as the tailored add-ins may do.

The first research question of the thesis is finding an efficient way to bundle a complex web application to be effortlessly used by third party developers in external sites. The proposed solution is a web component created with Lit that combines script-based embedding and an iframe.

The solution is bundled to a single script file that can be included to external sites with a single line of code. Using the solution to embed M-Files web client is effortless with a custom HTML element. The element can be configured by site administrators with attributes, similar to other HTML elements. Configuring the solution is possible for end users without technical knowledge. The custom element includes a configuration panel next to the web client that can be used to configure the M-Files web client without changes to the code. Site administrators and end users alike are able to configure and reconfigure the solution with minimal effort compared to the approaches before the solution.

The easy configuration is a result of the script-based solution where the configuration can be passed to easily from the host page. The script-based solution also enables the

creation of a configuration panel outside the complex web client. No changes are needed to the complex web application itself in the case of M-Files. The iframe isolates the complex web client. The web client only accepts predefined communications and maintains its internal state outside the hosting page. The downside is the communication with an iframe being more complicated than script-based communication. The created solution abstracts this communication and configuration to easily accessible element attributes and a visual configuration panel.

The thesis answers to the question of how suitable are existing frameworks for the effortless embedding approach. The thesis compared different frameworks on their suitability. The criteria compared for the frameworks are: the effort required for embedding the result, the longevity, browser compatibility and flexibility of the framework. Of the frameworks compared Lit and Stencil are deemed as feasible frameworks for implementing the generic add-in. Both frameworks create a standard web component and are compatible on modern and legacy browsers. Support for older browsers is added with transpiling and polyfills in the implemented prototype. The thesis chose Lit as the framework because of its better flexibility with full control over the bundling and other tools added to it. Stencil is a good alternative if extensive control over the build tools are not needed and a configuration free out-of-the box experience is preferred.

The thesis cannot pinpoint with absolute certainty the *best* method to package a complex web client to be effortlessly embeddable. The frameworks chosen for the comparison are only a fraction of the frameworks available. It is possible a better framework exists or that an even better approach than the combination of script-based and iframe embedding exists. Regardless of these limitations the thesis finds the chosen approach and framework to fulfil the objectives set for the thesis.

The solution is considered a success for embedding a complex web application effortlessly. Using the proposed method of including a complex web client in an iframe and abstracting the configuration and embedding to a script-based approach can be applied to other complex web applications. The method is beneficial when the application is complex to configure otherwise. If the application does not require complex configuration other solutions should be considered.

REFERENCES

[1] Li, Y.-F., Das, P. K. and Dowe, D. L. Two decades of Web application testing—A survey of recent advances. eng. *Information systems (Oxford)* 43 (2014), pp. 20–54. ISSN: 0306-4379.

[2] Vinegar, B. and Kovalyov, A. *Third-Party JavaScript*. eng. New York: Manning Publications Co. LLC, 2013. ISBN: 1617290548.

[3] Deveria, A. *Browser usage table*. URL: https://caniuse.com/usage-table (visited on 03/20/2022).

[4] Wirfs-Brock, A. and Eich, B. JavaScript: the first 20 years. eng. *Proceedings of ACM on programming languages* 4.HOPL (2020), pp. 1–189. ISSN: 2475-1421.

[5] Harband, J. *Standard ECMA-262, 12th edition / ECMAScript® 2021 Language Specification*. ECMA International. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-262_12th_edition_june_2021.pdf (visited on 03/20/2022).

[6] Wirfs-Brock, A. *Standard ECMA-262 6th Edition / ECMAScript® 2015 Language Specification*. ECMA International. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf (visited on 01/05/2022).

[7] Engelschall, R. S. *ECMAScript 6 - New Features: Overview & Comparison*. URL: http://es6-features.org/ (visited on 01/05/2022).

[8] Simpson, K. *You don't know JS : ES6 and beyond*. Sebastopol, CA : O'Reilly, 2016. 278 p.

[9] Mozilla Contributors. *Classes*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes (visited on 02/02/2022).

[10] *Class field declarations for JavaScript*. ECMA International. URL: https://github.com/tc39/proposal-class-fields (visited on 02/02/2022).

[11] Mozilla Contributors. *Arrow function expressions*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions (visited on 01/18/2022).

[12] Rauschmayer, A. *Exploring ES6*. URL: https://exploringjs.com/es6/ch_modules.html#sec_advantages-es6-modules (visited on 01/31/2022).

[13] Mozilla Contributors. *JavaScript modules*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (visited on 01/31/2022).

[14] Branscombe, M. *JavaScript Standard Moves to Yearly Release Schedule; Here is What's New for ES16*. URL: https://thenewstack.io/whats-new-es2016/ (visited on 03/11/2022).

[15] *The TC39 Process*. ECMA International. URL: https://tc39.es/process-document/ (visited on 03/11/2022).

[16] Zaytsev, J. *ECMAScript compatibility table*. URL: https://kangax.github.io/compat-table/es6/ (visited on 03/19/2022).

[17] *Firefox 78 Release Notes*. Mozilla Foundation. URL: https://www.mozilla.org/en-US/firefox/78.0/releasenotes/ (visited on 03/22/2022).

[18] Ilyushin, E. and Namiot, D. On source-to-source compilers. *International Journal of Open Information Technologies* (2016). URL: https://cyberleninka.ru/article/n/on-source-to-source-compilers (visited on 08/20/2021).

[19] *Transcrypt*. Python Software Foundation. URL: https://pypi.org/project/Transcrypt/ (visited on 09/13/2021).

[20] *TypeScript*. Microsoft Corporation. URL: https://github.com/microsoft/TypeScript (visited on 09/13/2021).

[21] Lunnikivi, H., Jylkkä, K. and Hämäläinen, T. Transpiling Python to Rust for Optimized Performance. eng. *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Vol. 12471. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 127–138. ISBN: 9783030609382.

[22] Sharp, R. *What is a Polyfill?* Oct. 8, 2010. URL: https://remysharp.com/2010/10/08/what-is-a-polyfill (visited on 09/13/2021).

[23] Mozilla Contributors. *String.prototype.startsWith()*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith (visited on 09/14/2021).

[24] Satrom, B. *Building Polyfills: Web Platform APIs for the Present and Future*. eng. Sebastopol: O'Reilly Media, Incorporated, 2014. ISBN: 9781449370732.

[25] *Definition of embed*. Oxford University Press. URL: https://www.lexico.com/definition/embed (visited on 02/10/2022).

[26] Mozilla Contributors. *<img>: The Image Embed element*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img (visited on 02/10/2022).

[27] Mozilla Contributors. *Multimedia and Embedding*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/ (visited on 02/10/2022).

[28] Mozilla Contributors. *<iframe>: The Inline Frame element*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe (visited on 02/10/2022).

[29] Mozilla Contributors. *Same-origin policy*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on 02/10/2022).

[30] Mozilla Contributors. *Web APIs*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/API (visited on 02/10/2022).

[31] Mozilla Contributors. *Quirks Mode and Standards Mode*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Quirks_Mode_and_Standards_Mode (visited on 03/19/2022).

[32] Mozilla Contributors. *Cross-Origin Resource Sharing (CORS)*. Mozilla Foundation. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (visited on 03/19/2022).

[33] *What is Salesforce?* Salesforce.com, Inc. URL: https://www.salesforce.com/products/what-is-salesforce/ (visited on 03/24/2022).

[34] *CRM 101: What is CRM?* Salesforce.com, Inc. URL: https://www.salesforce.com/crm/what-is-crm/ (visited on 03/24/2022).

[35] *Salesforce Lightning Component Framework*. Salesforce.com, Inc. URL: https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/intro_framework.htm (visited on 03/24/2022).

[36] *What is SharePoint?* Microsoft Corporation. URL: https://support.microsoft.com/en-us/office/what-is-sharepoint-97b915e6-651b-43b2-827d-fb25777f446f (visited on 03/24/2022).

[37] *Using web parts on SharePoint pages*. Microsoft Corporation. URL: https://support.microsoft.com/en-us/office/using-web-parts-on-sharepoint-pages-336e8e92-3e2d-4298-ae01-d404bbe751e0 (visited on 03/24/2022).

[38] *SharePoint Add-ins*. Microsoft Corporation. URL: https://docs.microsoft.com/en-us/sharepoint/dev/sp-add-ins/sharepoint-add-ins (visited on 03/24/2022).

[39] *M-Files*. URL: https://www.m-files.com (visited on 03/27/2022).

[40] Uitto, J. Taking code analysis into use for existing codebase. MA thesis. Tampere University, 2020.

[41] *M-Files Add-in for Teams and SharePoint Online - Installation Guide*. M-Files. URL: https://kb.cloudvault.m-files.com/vnext/#/vault/%7B3ECA226F-7B54-428B-B539-DE443E6134EC%7D/objects/DDA2961C-552D-4983-9943-7EB0A41B973E/latest (visited on 05/17/2022).

[42] *Lit website*. Google LLC. URL: https://lit.dev/ (visited on 05/07/2022).

[43] Amarasinghe, S. *Service worker development cookbook : build highly available and performant native web applications that seamlessly integrate with third-party APIs*. eng. Quick answers to common problems. Birmingham, England: Packt Publishing, 2016 - 2016. ISBN: 1-78646-952-9.

[44] *Lit GitHub page*. Google LLC. URL: https://github.com/lit/lit/ (visited on 05/07/2022).

[45] Potter, J. *Npmtrends npm package downloads analytics*. URL: https://www.npmtrends.com/@stencil/core-vs-lit-vs-lit-element-vs-lit-html (visited on 05/07/2022).

[46] Lubański, D. *Hybrids website*. URL: https://hybrids.js.org/ (visited on 05/18/2022).

[47] Lubański, D. *Hybrids GitHub page*. URL: https://github.com/hybridsjs/hybrids (visited on 05/18/2022).

[48] *Snuggsi GitHub page*. URL: https://github.com/devpunks/snuggsi (visited on 05/07/2022).

[49] *Snuggsi website*. URL: https://snuggsi.com/ (visited on 05/07/2022).

[50] *Stencil website*. Ionic. URL: https://stenciljs.com/ (visited on 05/18/2022).

[51] *Stencil GitHub page*. Ionic. URL: https://github.com/ionic-team/stencil (visited on 05/18/2022).

APPENDIX A: M-FILES GENERIC ADD-IN

```
1  import { html, LitElement } from 'lit';
2
3  import style from '../style/stylesheet.css';
4
5  export class MFiles extends LitElement {
6    // Styles are scoped to this element.
7    // They do not conflict with styles of the host page
8    // or other components.
9    static get styles() {
10     return style;
11   }
12
13   static get properties() {
14     return {
15       // Configuration panel properties.
16       serverURI: { type: String },
17       vaultGUID: { type: String },
18       viewURI: { type: String },
19       keyword: { type: String },
20       editing: { type: Boolean },
21
22       // Properties for internal use.
23       _integratorLoaded: { state: true, type: Boolean },
24       _loading: { state: true, type: Boolean },
25       _configOpen: { state: true, type: Boolean },
26     };
27   }
28
29   constructor() {
30     super();
31     this.serverURI = "https://www.mydomain.com";
32     this.vaultGUID = "00000000-0000-0000-0000-000000000000";
```

```
33      this.viewURI = "";
34      this.keyword = "";
35      this.editing = false;
36
37      this._integratorLoaded = false;
38      this._loading = true;
39      this._configOpen = true;
40    }
41
42    // Connected to the document.
43    connectedCallback() {
44      super.connectedCallback();
45      this.loadConfiguration();
46    }
47
48    // A configuration panel to enable modifying
49    // the settings from a user interface.
50    configurationPanelTemplate() {
51      return html`
52        ${this._configOpen ? html`
53          <div class="container">
54
55            <div class="wide flex" id="titlediv">
56                <h2 class="text title">M-Files Add-in</h2>
57                ${this._loading ? "" : html`
58                <span type="button" id="close" class="close"
59                  @click=${this.__configPanelVisibilityHandler}></span>
60                `}
61            </div>
62
63            <div id="inputgroup">
64
65              <div class="text" id="serversettings">Server Settings</div>
66
67              <div>
68                <label for="serverURI" class="text label">
69                  Server URI *
70                </label>
71                <div>
72                  <input type="text" id="serverURI"
```

```
73                    class="text textinput"
74                    @change=${this.__serverURIHandler}
75                    value="${this.serverURI}">
76                </div>
77              </div>
78
79              <div>
80                <label for="vaultGUID" class="text label">
81                  Vault GUID *</label>
82                <div>
83                  <input type="text" id="vaultGUID" class="text textinput"
84                    @change=${this.__vaultGUIDHandler}
85                    value="${this.vaultGUID}">
86                </div>
87              </div>
88
89              <div>
90                <label for="viewURI" class="text label">
91                  View URI (optional)</label>
92                <div>
93                  <input type="text" id="viewURI" class="text textinput"
94                    @change=${this.__viewURIHandler}
95                    value="${this.viewURI}">
96                </div>
97              </div>
98
99              <div>
100               <label for="keyword" class="text label">
101                   Keyword (optional)
102               </label>
103               <div>
104                 <input type="text" id="keyword" class="text textinput"
105                   @change=${this.__keywordHandler}
106                   value="${this.keyword}">
107               </div>
108             </div>
109
110             <div>
111               <label for="editing" class="text label">Editing</label>
112               <div>
```

```
113                 <label class="switch">
114                   <input type="checkbox" id="editing"
115                     @change=${this.__editingHandler}
116                     value="${this.editing}"
117                     ?checked=${this.editing}>
118                   <span class="slider round"></span>
119                 </label>
120                 <label for="editing" class="text">
121                   ${this.editing ? "Enabled" : "Disabled"}
122                 </label>
123               </div>
124             </div>
125
126           </div>
127
128           <div id="inputapply">
129             <button class="text" id="apply"
130         @click=${this.__applyHandler}>Apply</button>
131           </div>
132         </div>
133         ` : "" }
134     `
135   }
136
137   // Main content with the web client in an iframe.
138   contentTemplate() {
139     return html`
140       ${this._loading ? html`
141         <div class="center" id="loader"></div>` : "" }
142
143       <div id="iframe"
144             style="display:${this._loading ? "none" : "block"}">
145         <iframe id="vnext" width="100%" height="100%"
146           scrolling="no" @load=${this.__clientReadyHandler} src="">
147         </iframe>
148         ${this._configOpen ? "" : html`
149           <button id="reopen"
150             style="right:${this.editing ? "140px" : "107px"};"
151             @click=${this.__configPanelVisibilityHandler}>
152               &#9881;
```

```
153             </button>
154           `}
155         </div>
156       `
157     }
158
159     // Server URI changed.
160     __serverURIHandler(e) {
161       this.serverURI = e.target.value;
162     }
163
164     // Vault GUID changed.
165     __vaultGUIDHandler(e) {
166       this.vaultGUID = e.target.value;
167     }
168
169     // View URI changed.
170     __viewURIHandler(e) {
171       this.viewURI = e.target.value;
172     }
173
174     // Keyword changed.
175     __keywordHandler(e) {
176       this.keyword = e.target.value;
177     }
178
179     // Editing mode changed.
180     __editingHandler(e) {
181       this.editing = !this.editing;
182     }
183
184     // Configuration is confirmed.
185     __applyHandler(e) {
186       this._loading = true;
187
188       this.storeConfiguration();
189       this.applyConfiguration();
190     }
191
192     // Configuration panel is opened or closed.
```

```
193   __configPanelVisibilityHandler(e) {
194     this._configOpen = !this._configOpen;
195   }
196
197   // The iframe content has loaded.
198   __clientReadyHandler(e) {
199     let iframe = e.target;
200
201     // Don't do anything if no configuration exists.
202     if( this.serverURI === "https://mydomain.com" ||
203         this.vaultGUID === "00000000-0000-0000-0000-000000000000" ) {
204       return;
205     }
206
207     // Redirect to the initialization page if not visited yet.
208     if(this._integratorLoaded === false) {
209       iframe.src = this.serverURI + "/webuiintegrator.html";
210       this._integratorLoaded = true;
211       return;
212     }
213
214     // Disable the loading screen.
215     this._loading = false;
216
217     let view = "";
218     // Ensure vault GUID is available and prepare the default view path.
219     if( this.vaultGUID ) {
220       // Base view.
221       view = "#/vault/{"+ this.vaultGUID +"}";
222
223       // If default view set.
224       if( this.viewURI ) {
225         view += "/"+ this.viewURI;
226
227       // If keyword set.
228       } else if( this.keyword ) {
229         view += "/search?query="+ encodeURIComponent( this.keyword );
230       }
231     }
232
```

```
233   // Configuration of the web client.
234     const config = [
235       this.serverURI,
236       this.editing,
237       view
238     ];
239
240   // Send the configuration to the web client.
241   iframe.contentWindow.postMessage(
242     config, this.serverURI
243   );
244   }
245
246   // Load configuration from local storage.
247   loadConfiguration() {
248     const serverURI = localStorage.getItem("ServerURI");
249     const vaultGUID = localStorage.getItem("VaultGUID");
250     const viewURI = localStorage.getItem("ViewURI");
251     const keyword = localStorage.getItem("Keyword");
252     const editing = localStorage.getItem("Editing");
253
254     if(serverURI !== null) this.serverURI = serverURI;
255     if(vaultGUID !== null) this.vaultGUID = vaultGUID;
256     if(viewURI !== null) this.viewURI = viewURI;
257     if(keyword !== null) this.keyword = keyword;
258     if(editing !== null) this.editing = (editing === "true");
259   }
260
261   // Save configuration to local storage.
262   storeConfiguration() {
263     localStorage.setItem("ServerURI", this.serverURI);
264     localStorage.setItem("VaultGUID", this.vaultGUID);
265     localStorage.setItem("ViewURI", this.viewURI);
266     localStorage.setItem("Keyword", this.keyword);
267     localStorage.setItem("Editing", this.editing);
268   }
269
270   // React to user applying a configuration from the configuration panel.
271   applyConfiguration() {
272     this._integratorLoaded = true;
```

```
273
274       let iframe = this.renderRoot.querySelector("#vnext");
275       iframe.src = this.serverURI + "/webuiintegrator.html";
276     }
277
278     render() {
279       return html`
280         <div id="main">
281           ${this.contentTemplate()}
282           ${this.configurationPanelTemplate()}
283         </div>
284       `;
285     }
286   }
```

Program A.1. *M-Files generic add-in.*