Vrije Universiteit Amsterdam


Universiteit van Amsterdam

Master Thesis

# Dynamic Plugin System For Web Applications

**Author:** Gamma A. Octafarras

*1st supervisor:* Jaap Gordijn
*2nd reader:* Roel Wieringa

*A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science*

October 15, 2024

# Abstract

As the community of developers becomes more populated, there needs to be a convenient system for individual developers to contribute to an existing application without editing its codebase. A dynamic plugins system creates an ecosystem where multiple developers can easily collaborate to add functionality to an application. With such a system, it would be possible for these applications to have *plug and play* features. In this paper, we aim to create a dynamic plugin system that supports multiple front-end frameworks, mainly how the plugins are loaded into the host application and how it can be utilized to make the plugins in different front-end frameworks. We also create demo plugins using multiple front-end frameworks, which we then evaluate performance by running a web auditing tool. We also conduct a usability analysis through qualitative interviews with professionals. With the analysis, we see what type of plugins the specific front-end framework is suitable for.

# Contents

# List of Figures

## LIST OF FIGURES

# List of Tables

## LIST OF TABLES

# 1

# Introduction

The nature of software applications is that it keeps on continuing to evolve, whether it be in the sense of changes to previous features or the addition of features themselves. In traditional cases, an addition of a feature would require developers to edit the original codebase and further add more files. This would usually require the entire system to have some kind of recompilation and a quick restart. Although nowadays, modern practices such as continuous integration alleviate this by automating builds (1) this requires contributors to have significant knowledge of the existing systems. Moreover, bugs can become more prevalent in the traditional systems (2) as by adding more code to the codebase can inherently create unwanted interactions, which will need further extensive testing to find out and then eventually fix. A method to append isolated features to an application is then needed, and one of the methods to do that is the plugin system.

Fundamentally, a plugin system leverages multiple smaller applications to add or modify features to a larger application. The larger application could be the *main application* or there could be several larger applications. For the plugins to be able to be created, the larger application themselves need to be programmed to be able to support the plugins.

A plugin system also allows for easier collaboration with other users, in the sense that someone that has not partaken in the project itself can easily contribute by creating their own plugin. Moreover, as the plugins themselves have different options for the framework to be used, plugin creators do not have to have prior knowledge of the existing system, they can choose a framework they are comfortable with.

In this thesis, we will work towards answering what dynamic plugin systems are. how they are created and how useful they are in practice. These questions will be elaborated on further in Chapter 3.

## 1.1 Plugin Architecture

A general plugin architecture design would be a system with two main components: i) the core system and ii) the plugin components. These two components would then work with each other to allow the additional features in the form of plugins to be added to the main application. The key features of the design are to add extensibility, flexibility, and isolation of application features and customs processing logic. With the addition of a plugin system, the output should be a system where a plugin could be added or removed without interrupting the main application

In the case of this thesis, as the aim is to create a dynamic loading application, an additional layer that handles loading and installation is added.

### 1.1.1 Core System

The Core System, at a high level, would be some sort of an *integration* system, where it would integrate the plugin components to the main application. The core system should not be able to, in any way, have its functions altered by plugins. It should solely be an integrator. Additionally, the core system should be able to identify what and where the required plugins are and declare extension points that plugins can hook into. These extension points, these hooks, often represent the core system life cycle. As such, each plugin registers itself to the core. Each plugin registers itself to the core, passing some information such as handlers, data format, and hooks into these extension points.

### 1.1.2 Plugin Components

Plugin components are individual, custom applications that add functionality to the main application. These components are independent by nature and do not necessarily need knowledge of the parent application, other in the case of the plugin itself being an application that alters the main application. Components are basically units of composition that may be subject to composition by third parties(3).

There also exist plugins that require communication with other separate plugins. However, it is a good practice to keep code isolation and have as minimal communication from each other.

### 1.1.3 Dynamic Loading

The dynamic loading layer represents a method to dynamically load and install the individual plugins, without having the core system perform any kind of re-compilation or

re-initialization(4). A general way that this is done is to have a separate application that handles these tasks. The separate application in turn must know both the Core System and the Plugin Components and have some kind of runtime application shell for it to register the individual Plugin Components to the Core System dynamically.

## 1.2 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 discusses the existing work in the area of dynamic component-based applications and further creates a definition of what a dynamic plugin framework is. Chapter 3 defines the problem statement and the approaches taken by the thesis. In Chapter 4 we develop a specification of the micro-component architecture and discuss a dynamic plugin framework. Then, in Chapter 5 we evaluate and assess the results of the previous chapter. Chapter 6 discuss the results of the thesis and its implications of it for current and future research and practitioners. Chapter 7 summarises the contributions of this work and gives some suggestions for possible future extensions.

# 2

# Dynamic Plugin Systems

In this section, we research existing plugin systems and front-end loading techniques along with related literature.

## 2.1 Existing Plugin Systems

### 2.1.1 Atlassian Plugins

One of the most popular full-fledged plugin system is the Atlassian Marketplace(5). The Atlassian Marketplace houses plugins that they call as *apps* which can be added to their existing products such as Jira, Confluence, etc.

The plugin system that Atlassian created utilizes OSGi and are based on the three components (Figure 2.1):

1. **Services** Each bundle can publish and find interfaces and therefore use functionality from other bundles. JIRA itself is composed of some OSGi bundles itself. Therefore the plugins need to tell the Atlassian SDK which JIRA interfaces we want to use.

2. **Life-Cycle** Bundles can be installed, started and stopped during runtime. That means, while JIRA is running plugins can be installed, started and stopped.

2. **Bundles** Bundles (also called modules) only share Interfaces with other modules. You want to hide your implementation.

### 2.1.2 OpenHAB

OpenHAB is a home automation software where it contains a plugin system to add *add-ons*. Plugins are mostly integrated in the form of widgets, a simple component which

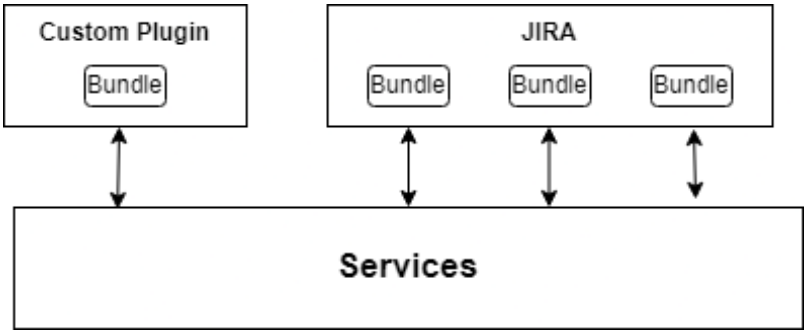


**Figure 2.1:** Atlassian JIRA Plugin System

aims to show users a single, focused piece of information. The components are mostly simple and editable UI elements, not complex module-in-module components. The system is more focused on having a dashboard which contains multiple simple widgets instead of full-fledged multifunctional plugins in separated pages.

### 2.1.3 LifeRay

Liferay is an open source enterprise web platform for building business solutions which has a plugin system for users to easily add-on features. Similar to Atlassian, plugins are distributed in a marketplace and are in the form of *Apps*. Apps can be complex and contain their own dashboard which may also contain more plugins in the app itself. LifeRay apps follow the module-in-module plugin design. However smaller plugins that offer functionalities (utility plugins) with no UI components are also supported.

## 2.2 Micro Front-ends

As this thesis mostly focuses on the front-end aspects of a plugin system. Several methods of composition and integration of Micro Front-ends are used. In this section we discuss previous studies on Micro Front-ends.

As Microservices become more prominent in the scaleable software development industry, extending the concept of the architecture to the frontend has become the approach that companies such as Allegro, HelloFresh, OpenTable, Skyscanner, DAZN and others (6, 7, 8, 9, 10, 11, 12) utilize as either their main frontend architecture or is used in some of their products.

The term micro frontend(MFE) gained traction around at 2016 (10, 13) in the industry as developers who were working on large scale projects had difficulties maintaining the

multiple front end features as a single codebase and were looking for an approach that would ease the management of separate features. As they were already adopting the micro service architecture in the back end, extending the concept to the front end was seen as a possible solution to avoid a large monolithic front end. An early definition of micro frontend (13) was: '*A web application broken up by its pages and features, with each feature being owned end-to-end by a single team. Multiple techniques exist to bring the application features—some old and some new—together as a cohesive user experience, but the goal remains to allow each feature to be developed, tested and deployed independently from others.*' The main ideas of the Micro Frontend Architecture can be formulated as:

- Technology Agnostic
- Code Isolation
- Native Browser Features for Communication
- Resilient Application

### 2.2.1 General Design approaches

A general design of MFE is to have one main container, usually composed by an *orchestrator*, which houses the multiple MFE applications. The main container may also contain base shared components such as the navigation menu and titles. As one of the main goals in adopting MFE is for each app to be independent, each app is isolated from each other. As depicted in Figure 2.2, each application is technologically agnostic where each app can run different frameworks. This general design is what makes MFEs scale well, as each application is independent they can be developed individually without blocking other applications.

With different the separate applications built, there needs to be a system to combine the different applications into the main container. This is usually done by a composition system(14) which will be discussed in the next section.

### 2.2.2 Composing methods

#### 2.2.2.1 Client side

In client-side composition (Figure 2.3),the application shell loads Micro-Frontends inside itself in the browser. MFEs should have as an entry point a JavaScript or HTML file so

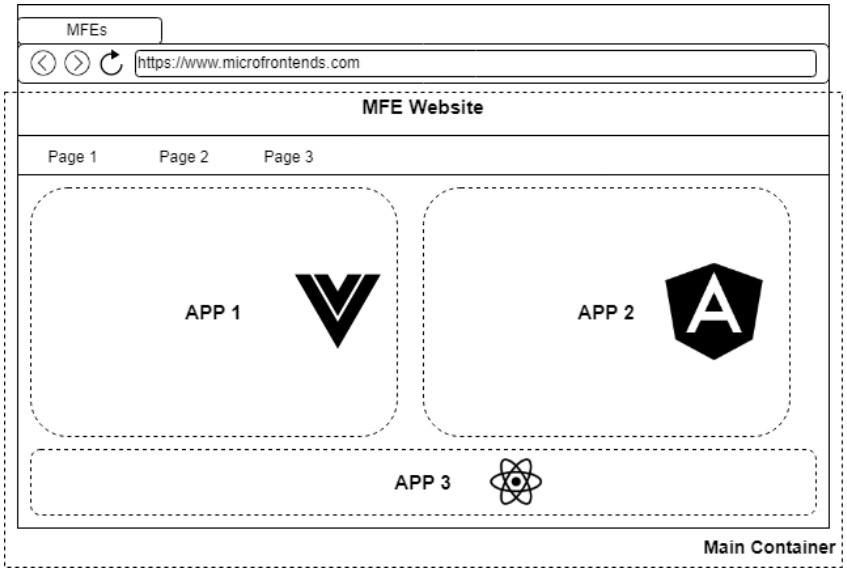


**Figure 2.2:** Basic Design of MFE

that the application shell can dynamically append the DOM nodes in the case of an HTML file or initialize the JavaScript application when the entry point is a JavaScript file. (14)

Another approach that is less popular is to utilize a combination of *iframe* HTML element, with each element hosting an MFE, a method utilized by Spotify (12) in both their desktop app and website.

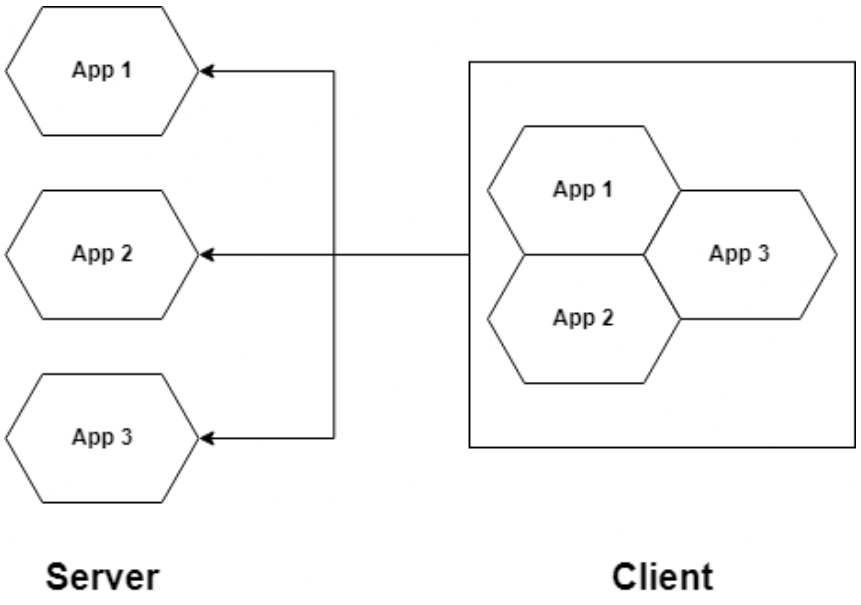


**Figure 2.3:** Client side composition

#### 2.2.2.2 Server side

In server-side composition, composition could happen either at runtime or at compile time (Figure 2.4). In this case, the server is composing the view by retrieving all the different MFEs and assembling the final page in the server-side before sending it to the client. This solution is a decent option if all users see the same page. However if the pages are personalized per user, it will require serious considerations regarding the scalability of the eventual solution as there will be multiple requests to compose different pages in the server.

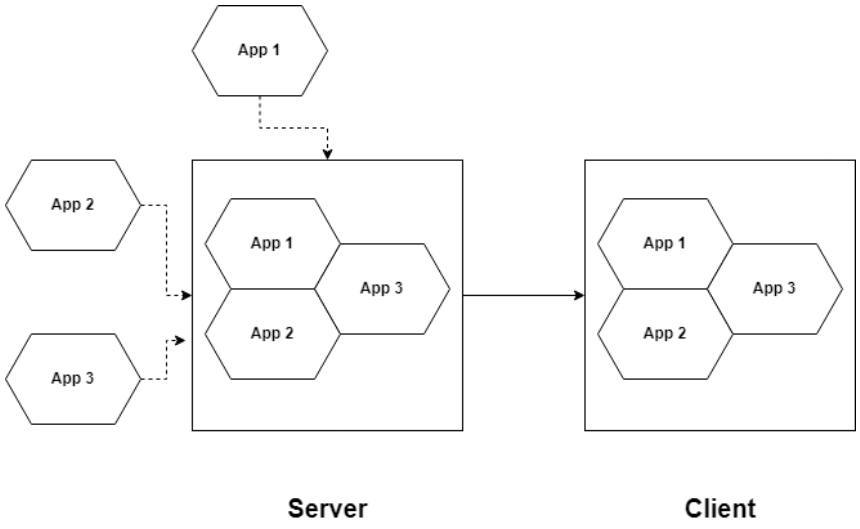


**Figure 2.4:** Server side composition

# 3

# Research Questions and Approach

In this section, we will discuss the various problem statements that we have formulated for this thesis and further define the approaches that will be taken for each individual problem.

## 3.1 Research Questions

The problem statements that we have used can be formulated as the following research questions:

*RQ1: What is a dynamic plugin system?*

*RQ2: How to create a dynamic plugin system that supports multiple front-end frameworks?*

*RQ3: How to use the dynamic plugin system for building a web application that supports plugins?*

*RQ4: Which front-end framework is ideal for building plugins?*

The three research questions defined above and the approaches will be further elaborated in the following sections.

### 3.1.1 RQ1: What is a dynamic plugin system?

This research question is formulated to get the background and basic concepts of what dynamic plugin systems are. For this question, we will conduct a systematic multivocal literature review encompassing the subjects of plugin system and micro front-ends.

The Multivocal Literature Review (MLR)(15) process will be used in this study as Micro Frontends (MFEs) and the dynamic plugin system is a relatively recent concept and has only recently gained traction as web applications continue to grow in size. Hence there is a lack of academic research in the field, however many companies in the industry have experimented and adopted MFEs. With several publishing experience reports and their findings as *grey* literature, which makes the systematic MLR the ideal process for this specific topic. In this MLR, we will include both published papers and *grey* literature, which in this literature review will include tech blogs, experience reports and online talks(podcasts/videos) with the reasoning that the reports of these companies can be a basis for future research on the subject.

This research question is discussed in Chapter 2.

### 3.1.2 RQ2: How to create a dynamic plugin system that supports multiple front-end frameworks?

This research question is formulated to give an understanding of how dynamic plugin systems are created as a framework and how they also are able to support multiple frameworks for the front-end components.

For this research question, we extended an existing dynamic plugin framework called *micro-components* [1] to i) support multiple operating systems for development and ii) support multiple front-end frameworks.

This research question will be discussed in Chapter 4.

### 3.1.3 RQ3: How to use the dynamic plugin system for building a web application that supports plugins?

This research question is formulated to assess and evaluate the dynamic plugin framework in practice and to assess if they are truly framework agnostic.

The approach that we have taken for this question is to create two web applications. The first application will be used as the *host application* which we will then attach to the dynamic plugin framework. The second application will be the plugin itself, for this thesis we created a file browser plugin for the sake of evaluation. To test if the framework is truly front-end agnostic, we created the plugin with two different frameworks and assess the similarity and/or differences.

This research question will be discussed in Chapter 5.

[1] https://github.com/corneliusoficu/micro-components

### 3.1.4 RQ4: Which front-end framework is ideal for building plugins?

For this research question, an analysis with regards to performance and development process to the individual codebases were conducted. Further, to get a wider range of opinions, we also approached developers that have no experience in the frameworks used in the experiment to prevent bias. We then ran a qualitative interview with the developers to get their insights and opinion on the frameworks. The results of the analysis and the qualitative interviews will then be compiled into a table

This research question will be discussed in Chapter 5.

# 4

# A dynamic plugin framework

In this section, we discuss the *micro-component* framework and the extensions that we have done.

## 4.1 Micro-components

The system is based on the three layers of abstraction: i) Host Application ii) Karaf Runtime Middleware and iii) Micro-component Development and Deployment Environment. A custom CLI application will be the main method of composing and installing the plugins to the host application.

## 4.2 Host Application

A parent application that serves as the main front-end component. While the Host Application mainly hosts the deployed micro-component, it also has main shared functionalities. In the case of the example host application, the shared functionalities are the dashboard view.

The host application communicates with the Karaf Runtime layer in order to fetch the plugins that have been developed and built in the Micro-component layer. The host application then will load the plugins using methods that will be explained in the subsequent sections.

While the example host application is designed in the ways mentioned above, the host application is relatively flexible in the sense that future developers are able to design their own host front-end and still attach the other 2 layers without complex modifications.

### 4.2.1 Karaf Runtime Middleware

Front-end and back-end are created separately from this layer, but the base templates are bootstraped through this layer.

This layer acts as the application container the entire system. Main function of this layer is to run the backend of the host application and the integration of the micro-components with the host application. The environments that are required for both the micro-components and the backends are stored in this runtime. The databases are also stored in this layer.

The back-end and the front-end are then bundled into an OSGI project, which it then exports as a single bundle which in the case of this master project, will be called as the micro-components itself. This bundle is then loaded into karaf runtime.

### 4.2.2 Micro-component Development and Deployment Environment

In this layer, we created a custom CLI which mainly performs templating. The CLI generates a boiler-plate based on the front-end framework of choice which can be then used by the developers to develop the plugin.

### 4.2.3 Component Technologies

In this section, we will briefly explain the various technologies used in the framework itself to get a base idea on the component technologies.

#### 4.2.3.1 Java JAX RS

JAX-RS is a JAVA based API that supports the creation of RESTful Web Services. In this project the Java JAX RS was used as the back-end platform as it supports runtime in Karaf.

#### 4.2.3.2 MongoDB

MongoDB is used as the DB of both the individual plugins and the host application itself. MongoDB is a document-oriented NoSQL database that uses collections and documents instead of the traditional tables and rows. Documents are made out of key-value pairs and collections contain sets of documents which is the equivalent of tables in traditional relational databases.

#### 4.2.3.3 Vagrant

Vagrant is a software that we used for portable virtual environments. We opted to use this as it would simplify the development software configs and environments for other developers to use. Further, vagrant also supports multiple OS as long as the configurations are setup sufficiently.

## 4.3 Extensions

In addition to the previous dynamic plugin framework, we added extensions to the plugin to tailor it to the requirements of this thesis. Of which will be discussed in the following sections.

### 4.3.1 Multiple operating system development support

Due to the previous author developing the project on MacOS, the development setup hence was only provisioned to run on MacOS. One of the extensions that was done as part of this project was to make the development process operating system agnostic, where the project could be developed and setup in 3 major operating systems: Windows, MacOS and Ubuntu.

As the portable software environment Vagrant is used for this project, the extensions done were mostly on the vagrant configuration and setup files itself.

#### 4.3.1.1 File Syncing

Firstly, as vagrant fundamentally runs a VM for the development process, some kind of file syncing is necessary in order to develop seamlessly instead of having to either manually copy and paste code into the vm or doing some kind of version control syncing. Inherently, the file syncing in vagrant is not supported on multiple operating systems, an example being the default Network File System (NFS) not working on Windows Host. To alleviate this, we needed a way to mount the files using some kind of SFTP, which is one of the more possible methods as essentially vagrant runs a VM and most OSs have a method to SSH into VMS. This method requires the host to have an sftp-server installed, which is not a problem in MacOS or Ubuntu as it is usually pre-installed. For Windows host, the sftp-server would need to be installed separately.

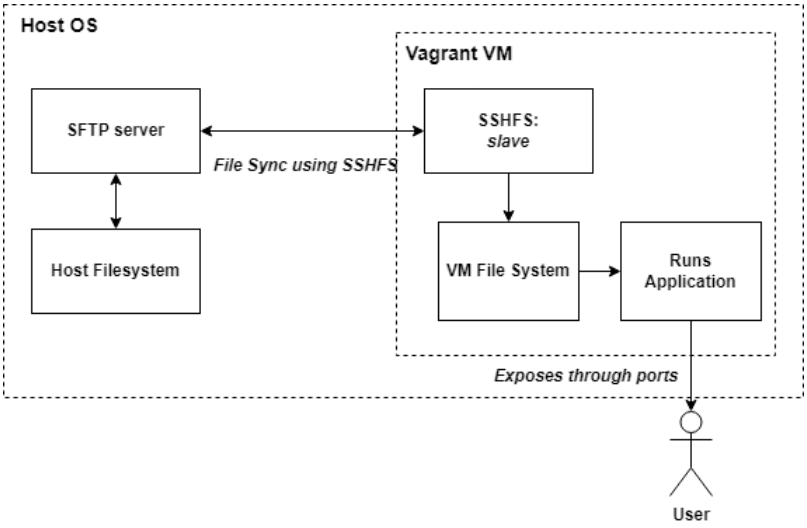


**Figure 4.1:** File Syncing Process

The vagrant extension *vagrant-sshfs*[1] is the closest match to the requirements we needed for the file syncing, which we therefore utilize. SSHFS essentially uses the sftp-server software that exists on the host and sshfs running in slave mode within the guest to create a connection using the existing authentication over SSH that vagrant sets up. As SSHFS creates a slave node within the VM, we also added the *vagrant-vbguest* to automatically install the VirtualBox guest addons. These additions are added to VagrantFile as seen in the code listing (4.1) below. The file syncing process is illustrated in Figure (4.1).

**Listing 4.1:** Vagrantfile; SSHFS and VBGuest addons

```
...
unless Vagrant.has_plugin?("vagrant-sshfs")
  puts 'Installing vagrant-sshfs Plugin...'
  system('vagrant plugin install vagrant-sshfs')
end
unless Vagrant.has_plugin?("vagrant-vbguest")
  puts 'Installing vagrant-vbguest Plugin...'
  system('vagrant plugin install vagrant-vbguest')
end
```

#### 4.3.1.2 DHCP Issues

As vagrant runs a VM, two network adaptors should be created in Virtualbox. The first should be attached to the NAT network and allow the machine out bound connectivity.

[1] https://github.com/dustymabe/vagrant-sshfs

The second adapter should be attached to the Host-Only adapter connected to the network in order to support the file syncing mentioned in the previous section. While the original setup seems to work on MacOS, we had DHCP conflict issues when working in Ubuntu and Windows. The cause of this appears to be that vagrant does not like custom networks used for the file syncing and gives an error in which it considers both networks to be the same hence the conflict.

This issue was resolved by creating a Monkeypatch work-around, shown in listing (4.2) , to work with DHCP conflict issues. This essentially forces vagrant to use existing networks instead of trying to create a new one that will be read as conflicting.

**Listing 4.2:** Vagrantfile; Monkeypatch work-around

```
class VagrantPlugins::ProviderVirtualBox::Action::Network
  def dhcp_server_matches_config?(dhcp_server, config)
    true
  end
end
...
```

### 4.3.2 Multiple front-end framework support

The next extension of the framework that was done for this thesis was the modification of the front-end creation process to allow multiple front-end frameworks. Further, as an example of the ability to add another front-end framework we added support for VueJS to the plugin framework.

To facilitate this there are three major additions that need to be done: i) Template creation of the front-end, ii) compilation and building of the front-end into a single JS file and iii) the loading of the plugin to the host application.

#### 4.3.2.1 Front-end Templating

Firstly in order to support multiple front-end frameworks, there first needs to be a way to generate a common template as a boilerplate. Although it is technically possible to not use a template, templating greatly reduces errors and ease of integration. For the addition of Vue support, a vue generator was added to the CLI application.

#### 4.3.2.2 Compilation and building

To be able to support multiple front-end frameworks, the plugin must build into a native HTML component instead of a framework-specific component. An example of one on a

Vue plugin can be seen in listing (4.3).

```
vue-cli-service build --target wc --name mc-plugin-name [file]
```

**Listing 4.3:** Vue single file build

One caveat of this method is that host applications and the plugins data-bindings will no longer be inherently compatible. However, this is a quality-of-life caveat as the data will still be accessible through native HTML property reads.

#### 4.3.2.3 Loading into Host Application

Originally, as the framework only supported Angular front-ends, the host application natively loads the JS of the plugin as an angular component. However, in the case of multiple front-end frameworks this is inherently not possible as data-bindings and framework specific features will cause conflicts and errors. The approach that we have used to facilitate this is to use native HTML iFrames, inspired by the method that Spotify uses(16). Since an iFrame is a native HTML element, this would make sure that this implementation would work in host application regardless of the framework. This method requires the plugins to build into a single JS file, which is then loaded via url by the host application as seen in figure 4.2.

With iFrames set as the plugin container, we then need of a way to load the plugins itself into the iFrame. Several methods of loading the plugins were tested while developing this plugin of which will be explained in the next section.

#### 4.3.2.4 Loading of Plugins into iFrame

Previously, as the framework was made only for Angular, the plugins were loaded using the Angular extension *ax-lazy-element*[1] as shown in listing 4.4. For this project, we had to change this to a method that ideally uses native JS functions to support multiple front-end frameworks.

**Listing 4.4:** Original loading method

```
<ax-lazy-element *axLazyElementDynamic="url: getPluginLoadUrl">
</ax-lazy-element>
```

**Listing 4.5:** Modified loading method

```
const getBlobURL = (code, type) => {
  const blob = new Blob([code], { type })
```

[1] https://angular-extensions.github.io/elements/home

```
    return URL.createObjectURL(blob)
}
```

The methods that we tried for loading the plugins into the iFrame:

1. **srcdoc tag** In this method, we directly inject the URL of the compiled plugins to the iFrame by using the srcdoc tag. This was the simplest approach but 2 main drawbacks. The first would be that there is the known bug where the html/css code of the plugin could be escaped to the host application with the *</script* or *</style>*. The second would be that srcdoc is not supported in multiple browsers such as edge and IE.

2. **Blob pseudofiles** In this method, we utilize the Blob constructor to create a pseudo-file of the plugins and load it into memory. Which we then use the *URL.createObjectURL(blob)* function to create a url for the loading of the blobs. This method essentially simulates a hosted web app which we would as normal to load by url. This was the approach that we used in the final iteration of the project.

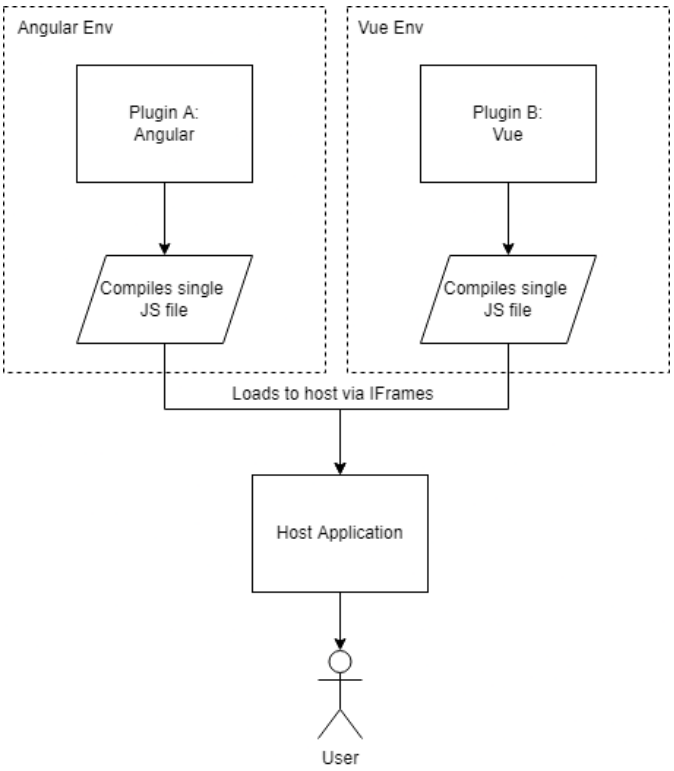


**Figure 4.2:** Multiple framework plugin loading

# 5

# Evaluation of the framework

In this section, we discuss how we evaluated the framework and analyze the results.

## 5.1 Evaluation Methodology

As a plugin framework is a tool that would be used by multiple people as a method of collaborating and/or contributing, we found it necessary for this project not to only evaluate the technical details of the framework but also its usability.

### 5.1.1 Performance Analysis

To test the performance of the individual plugins, we used the Lighthouse[1] tool. With lighthouse, the metrics that we focused on are:

- First Contentful Paint
  First Contentful Paint marks the time at which the first text or image is painted.

- Time to Interactive
  Time to interactive is the amount of time it takes for the page to become fully interactive.

- Speed Index
  Speed Index shows how quickly the contents of a page are visibly populated

- Largest Contentful Paint
  Largest Contentful Paint marks the time at which the largest text or image is painted

[1] https://github.com/GoogleChrome/lighthouse

### 5.1.2 Usability Analysis

Usability Evaluation focuses on how well users can learn and use a product to achieve their goals. It also refers to how satisfied users are with that process. For this analysis, we collected data from conducting multiple rounds of qualitative interviews with 6 Software Engineers and IT-related professionals seen in Table 5.1. To make sure that no framework bias occurs, we selected participants that has no to little experience in both Angular and Vue. The interviews were done through a mix of question-answer forms and follow up direct interviews which we conducted through a video call. Some of the interviews were done on a group basis with multiple professionals were interviewed together.

| Position | Company | Experience in front-end |
|---|---|---|
| Front-end Developer | DANA | Yes |
| Cybersecurity Engineer | EY | No |
| Game Developer | Independent | Yes |
| Front-end Intern | DANA | Yes |
| Java Developer | Software House | No |
| Back-end Lead | Spenmo | No |

**Table 5.1:** Interview Respondent List

## 5.2 Individual Plugins

Other than the host application we created 3 coherent plugins where all 3 are included in the Host Application. Further, we created each plugin in the 2 frameworks: Angular and Vue. We then first showed the respondents the codebase of the plugins followed by the compiled plugins. We then asked each respondents the following questions and their summarized answers are:

1. **What are the differences between the two codebases** The respondents observed that there are 2 major differences between the Vue and Angular codebases. The first being the file structure, with angular having significantly more files compared to the Vue counterparts and the writing of Logic and UI separation of Angular. The second being the language itself.

2. **What are the similarities?** For this questions the majority of respondents had an agreement where the base contents are similar. The code for the views are exactly

the same other than the data bindings. The CSS is also the same, just one being loaded externally and the other internally. However, one respondent had a different observation: "Even though the HTML and JS contents are basically the same, it can be said that it is because it was one developer that created both versions. If taken 2 developers, one more skilled in Vue and one in Angular, the resulting codebase for the same plugin could be different."

3. **As a dev contributing to a plugin which codebase would you use?** For this question, there was a mix of answers from the respondents. However there seems to be similar reasoning across all the respondents, in which they state that fundamentally, it would differ based on the factors (seen in table 5.2): i) project size, ii) number of people in the project and iii) time requirement. The respondents agreed that Angular would be the better choice if the plugin was a large project made by a team of developers but Vue would be better for smaller plugins made by a single developer.

   A direct quote from a respondent: "Even though the factors I said would differ my opinion on which framework to use, at the end of the day, even in a large project made by a group of developers, the comfortability of the devs is even a bigger factor."

The specific answers for each individual respondent can be seen in Appendix A.

| | Vue | Angular |
|---|---|---|
| Small Project | 5 | 1 |
| Large Project | 2 | 4 |
| 1 Dev | 5 | 1 |
| 2 Devs | 4 | 2 |
| >2 Devs | 2 | 4 |
| Long time requirement | 2 | 4 |
| Short time requirement | 5 | 1 |

**Table 5.2:** Framework Preferability

### 5.2.1 Performance of plugins

Other than the usability analysis, the performance of each individual plugin, both in Vue and in Angular was tested. Each test was run 5 times and its mean is used for the results, the individual results can be seen in Appendix B. The result of the performance test can be seen in the following table.

| Plugin | Vue | | | Angular | | |
|---|---|---|---|---|---|---|
| | Login | Register | Browser | Login | Register | Browser |
| First Contentful Paint | 0.7 | 0.7 | 0.9 | 0.7 | 0.8 | 0.8 |
| Time to Interactive | 1.9 | 1.9 | - | 1.9 | 2 | - |
| Speed Index | 1.4 | 1.4 | 1.8 | 1.4 | 1.6 | 1.8 |
| Largest Contentful Paint | 1.7 | 1.7 | - | 1.7 | 1.8 | - |

**Table 5.3:** Performance analysis

# 6

# Discussion

In this section, we discuss the our evaluations and put it into context with our research questions.

## 6.1 Dynamic Plugin Framework

In this project we extended an existing dynamic plugin framework to support multiple front-end framework. It can be said that the outcome of this project would a tool that is *front-end framework agnostic*. Along the way of achieving the desired results, we have also added development support for developing the framework on multiple operating systems. During the process, there were issues that we encountered during the setup of the project, which we discussed along with how we solved them in Chapter 4.3.1.

For the scope of this project, we developed plugins with 2 different front-end frameworks (Vue and Angular).

## 6.2 Evaluation

### 6.2.1 Performance

In our performance evaluation, it can be seen that while the performance of the 2 frameworks we used are similar, Vue has a very slight edge compared to Angular in terms of loading time. This result is somewhat to be expected as Vue utilizes a Virtual DOM, in which only updated parts of the UI are updated in comparison to Angular which directly manipulates the DOM. However, a limitation to this experiment is that the example plugins were all very small in size ($<$1500 lines of code), this might be the reason of why the performance between Vue and Angular only has small variations.

Another observation is that whilst the Angular components had larger file size, hence larger network payload while loading the plugin, the loading times are still very similar.

A future evaluation could test larger plugins and more frameworks.

### 6.2.2 Usability

As the output of this project is a tool that is used for creating dynamic plugins, the ideal use case would a collaborative project where multiple developers create the plugins. Hence we found it increasingly important to also conduct usability tests.

From the evaluation we conducted, we wanted to find out which framework out of the 2 tested would be the best for creating plugins. However, as seen in the evaluation it is a difficult task to determine which framework is, per se, *the best*. We can see from the result of interviews with professionals that there are different factors on which framework would be preferred. Further these factors, in a real-world situation, are usually a combination of the factors we listed in Section 5.2. There might be a case where a large project is done a single developer, where in that case, even the respondents hesitated to say which frameworks they preferred.

The limitation of the usability test also runs in hand with the performance evaluation. As this test was done with respondents with no experience in Angular or Vue, an interesting evaluation for the future would be to include respondents with experience in both frameworks as they might have deeper insights.

# 7

# Conclusions and Future Work

In the span of this project, we extended an existing dynamic plugin framework by: i) adding development support for multiple OS and ii) added support for multiple front-end framework.

The multiple OS development was implemented by packaging the framework itself with vagrant and subsequently adding the custom file-syncing extension *vagrant-sshfs*. Moreover, we presented the multiple techniques and isolated the ideal one for loading the plugins into the host application as the previous implementation was not suitable for multiple front-end frameworks. The addition of multiple front-end support was added by extending the templating, compilation functions in the Micro-component development layer of the framework.

We then developed multiple plugins using 2 different front-end frameworks and further evaluated by the two main criteria: i) Performance and ii) Usability. For the performance aspect, we utilized the web auditing tool Google Lighthouse to measure the various loading time of the components and for the usability, we conducted multiple rounds of interviews with professionals to get their insights and opinions based on the plugins we have developed.

With this project, we have added support for VueJS, but other front-end frameworks will be able to be added by future developers by adding a template.

A future work could look into eliminating or automating the framework-specific methods of templating and compilation of the front-ends rather than having the developers write framework-specific code for these for each framework that is added. Beyond that, further evaluations can be done with larger plugins and more frameworks to have less restricted results.

# References


[1] Mathias Meyer. **Continuous Integration and Its Tools**. *IEEE Software*, **31**(3):14–16, 2014. 1

[2] Jasmine Latendresse, Rabe Abdalkareem, Diego Elias Costa, and Emad Shihab. **How Effective is Continuous Integration in Indicating Single-Statement Bugs?** In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 500–504, 2021. 1

[3] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. 01 2002. 2

[4] Robert Chatley, Susan Eisenbach, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. **Predictable Dynamic Plugin Systems**. **2984**, pages 129–143, 03 2004. 3

[5] Tom Tsao. **The Atlassian Marketplace picks up momentum**, Dec 2021. 5

[6] Bartosz.walacik Bartosz.galek. **Managing frontend in the microservices architecture**, Mar 2016. 6

[7] Luca Mezzalira. **Adopting a micro-frontends architecture**, Apr 2019. 6

[8] Simon Tabor. **How DAZN manages micro-frontend infrastructure**, Aug 2021. 6

[9] Pepijn Senders. **Front-end microservices at Hellofresh**, Aug 2017. 6

[10] OpenTable. **OpenComponents - microservices in the front-end world**, Apr 2016. 6

[11] Jan Stenberg. **Experiences using micro frontends at IKEA**, Aug 2018. 6

## REFERENCES


[12] Author luca mezzalira. **Adopting a micro-frontends architecture**, Apr 2019. 6, 8

[13] **Micro frontends: Technology radar**, Nov 2016. 6, 7

[14] Caifang Yang, Chuanchang Liu, and Zhiyuan Su. **Research and Application of Micro Frontends**. *IOP Conference Series: Materials Science and Engineering*, **490**:062082, 04 2019. 7, 8

[15] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. **Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering**. *CoRR*, **abs/1707.02553**, 2017. 12

[16] Mostafa Biomee. **Micro fronends-spotify approach iframes (part 2)**, Aug 2020. 20

# Appendix A

# Qualitative Interview Answers

*Answers were not all in English and have been translated.

---

What are the differences between the two codebases?

1. The most obvious other than the language would be the way that the scripts are written. Structure of the files seems different as well, but it might just way the way it was made.

2. Looking at the script files, language is the most obvious one, one being TypeScript and the other JS. Structure is also clearly different, Angular has a clear cut structure. Some of the a-tags in the HTML is also different.

3. File layout. Angular has some kind of MVC-like structure with logic and view separated. Vue is just a single JS file with HTML CSS and JS combined. Data-bindings(?) are also different, more like language specific.

4. Both looks pretty similar to me if we only talk about the actual written code. So difference would be the things outside written code then, like file structure, testing etc.

5. Discounting framework-specific code and boiler plates, I'd say the JS code is where its different. Angulars https also needs some kind of setup it seems compared to Vues library import.

6. Attribute tags in the HTML is different ( ng-ifs & v-ifs) but thats framework specific. File structure is different too but maybe that's just how you made it. Language in the JS/TS file is also different.

What are the similarities?

1. The code for the views are exactly the same other than the data bindings. The CSS is also the same, just one being loaded externally and the other internally
2. If you translate the typecript to JS, probably around 70-80% of the code is the same. HTML and CSS is also exactly the same. But all this is probably because only 1 person wrote the code.
3. It seems like the code is exactly the same. I wouldn't count the framework specific code as different as that's something that web devs usually dont write.
4. Both look the same html wise. JS and the TS is also the same if you look at its logic. CSS can probably be shared between the too so its the same also.
5. HTML/CSS code is exactly the same if you dont count the data bindings and the conditionals in the HTML code.
6. Other than the framework-specific attribute tag, the HTMl code is basically the same.

As a dev contributing to a plugin which codebase would you use?

1. Its hard to say with comparing just the two as the codebase is very small. I've heard that vue is simpler and can see that the file structure is so, so maybe Vue for now.
2. Depends on the requirements, is it a larger project? I would prefer Angular for larger projects, the file structure seems more neat and easier to manage. Has built in unit testing too. And I prefer Typescript anyway so Angular.
3. If its only me writing the code, I'd say Vue. Vue's looks simpler just by glancing at the folder structure. The https fetches looks easier to incorporate.
4. It really depends. How long is the project requirement? Is it an individual project or a team project? How long am I given to make it?
5. I feel like the Angular structure is better, though the Vue looks smaller I imagine it can get messier as it grows? So probably Angular.
6. Vue. Although Angular's file structure seems more organized, you can probably do the same with the Vue code too. Writing small plugins I wouldn't need the extra out of the box angular features anyway too.

---

After looking at the codebase which framework do you generally prefer?(discounting the factors)

1. Vue
2. Angular
3. Vue
4. Vue
5. Angular
6. Vue

# A. QUALITATIVE INTERVIEW ANSWERS

# Appendix B

# Performance Tests

| Plugin | Vue | | | Angular | | |
|---|---|---|---|---|---|---|
| | Login | Register | Browser | Login | Register | Browser |
| First Contentful Paint | 0.7 | 0.7 | 0.9 | 0.6 | 0.8 | 0.8 |
| | 0.6 | 0.6 | 0.8 | 0.7 | 0.9 | 0.7 |
| | 0.6 | 0.6 | 0.9 | 0.7 | 0.7 | 0.8 |
| | 0.7 | 0.7 | 0.8 | 0.7 | 0.8 | 0.8 |
| | 0.7 | 0.7 | 0.9 | 0.6 | 0.8 | 0.8 |
| Time to Interactive | 1.9 | 1.9 | 2 | 1.9 | 2 | 2 |
| | 1.9 | 1.9 | 2.1 | 1.9 | 1.9 | 2 |
| | 1.9 | 1.9 | 2 | 1.9 | 2 | 2 |
| | 1.8 | 1.8 | 1.9 | 1.7 | 1.9 | 2 |
| | 1.8 | 1.8 | 1.9 | 1.8 | 2 | 1.9 |
| Speed Index | 1.5 | 1.5 | 1.9 | 1.4 | 1.6 | 1.8 |
| | 1.3 | 1.4 | 1.7 | 1.3 | 1.5 | 1.8 |
| | 1.4 | 1.3 | 1.8 | 1.4 | 1.6 | 1.8 |
| | 1.4 | 1.4 | 1.8 | 1.3 | 1.5 | 1.8 |
| | 1.4 | 1.4 | 1.8 | 1.4 | 1.6 | 1.9 |
| Largest Contentful Paint | 1.6 | 1.6 | 1.7 | 1.8 | 1.5 | 1.7 |
| | 1.6 | 1.6 | 1.8 | 1.7 | 1.8 | 1.7 |
| | 1.7 | 1.7 | 1.7 | 1.7 | 1.8 | 1.7 |
| | 1.7 | 1.7 | 1.8 | 1.6 | 1.9 | 1.9 |
| | 1.7 | 1.7 | 1.8 | 1.7 | 1.8 | 1.7 |

**Table B.1:** Performance analysis results