INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Analysis of Module Federation Implementation in a Micro-Frontend Application

DIOGO VALENTIM CARVALHO SOARES
novembro de 2023

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Analysis of Module Federation Implementation in a Micro-Frontend Application

Diogo Valentim Carvalho Soares (1171244)

Dissertation for obtaining the Degree of Master in Computer Engineering, Specialization Area in Software Engineering

Advisor: Paulo Gandra de Sousa

Abstract

This thesis explores the implementation of Module Federation in a micro-frontend application, a technique for dynamically loading components from various bundles at runtime. It is presented an overview of the micro-frontend architecture emphasizing its role in enhancing development efficiency and scalability. This thesis highlights the integration and potential challenges of Module Federation, focusing on providing optimal performance of the federated applications. The insights presented in this research provide a solid framework for developers and organizations aiming to leverage micro-frontends in conjunction with Module Federation in building efficient, scalable, and integrated web applications.

Keywords— Module Federation, Web Components, Micro-Frontend, Performance

Resumo

Esta dissertação explora a implementação de Module Federation numa aplicação Micro-Frontend, uma técnica usada para carregar dinamicamente componentes de vários pacotes em tempo de execução. É apresentada uma visão geral da arquitetura Micro-Frontend, enfatizando o seu papel no aumento da eficiência e escalabilidade do desenvolvimento. Esta tese destaca a integração e os possíveis desafios de Module Federation, focando em fornecer um desempenho ótimo para as aplicações. As percepções apresentadas nesta pesquisa fornecem um sólido quadro de referência para programadores e organizações que visam aproveitar Micro-Frontends e Module Federation na construção de aplicações web eficientes, escaláveis e integradas.

Keywords— Module Federation, Web Components, Micro-Frontend, Desempenho de Aplicações

Declaração de Integridade

Eu, Diogo Valentim Carvalho Soares, nº 1171244, aluno do Mestrado de Engenharia Informática do Instituto Superior de Engenharia do Porto, declaro ter atuado com absoluta integridade na elaboração deste documento. Nesse sentido, confirmo que NÃO incorri em plágio (ato pelo qual um indivíduo, mesmo por omissão, assume a autoria de um determinado trabalho intelectual ou partes dele). Mais declaro que todas as frases que retirei de trabalhos anteriores pertencentes a outros autores foram referenciadas ou redigidas com novas palavras, tendo neste caso colocado a citação da fonte bibliográfica.

Thanks to Professor Paulo Gandra, to the MyWorkplace team and especially to my family for their constant support and patience.

Contents

List of Figures

List of Tables

Listings

Notation and Glossary

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
BMW	*Bayerische Motoren Werke*
CDN	*Content Delivery Network*
CI/CD	*Continuous Integration/Continuous Delivery*
CJS	*CommonJS*
CLI	*Command-line Interface*
CLO	*Component Level Ownership*
CSR	*Client Side Rendering*
CSS	*Cascading Style Sheets*
CTW	*Critical TechWorks*
DOM	*Document Object Model*
ES6	*ECMAScript 6*
ESI	*Edge-side Includes*
ESM	*ECMAScript Modules*
GraphQL	*Graph Query Language*
gRPC	*Google Remote Procedure Call*
HTML	*Hypertext Markup Language*
iFrame	*Inline Frame*
IIFE	*Immediately-invoking function expressions*
LeSS	*Large-Scale Scrum*
MWP	*MyWorkplace*
NPM	*Node Package Manager*
REST	*Representational State Transfer*
SEO	*Search Engine Optimization*

SPA	*Single Page Application*
SSR	*Server-Side Rendering*
SWC	*Speedy Web Compiler*
UMD	*Universal Module Definition*
URL	*Uniform Resource Locator*

Chapter 1

Introduction

This chapter aims to provide an overview of the document, the current context of this work and explain the problem on which this dissertation is based. Subsequently, the objectives, approach, and defined development process will be presented.

1.1 Context

Critical TechWorks (CTW) is a company formed as a result of a partnership between BMW Group and Critical Software. This company was exclusively created to support BMW in development of software (Critical TechWorks, 2023). The MyWorkplace portal serves as an internal application integration solution, enabling access to multiple internal tools through a single web portal. This highly adaptable platform is not only widely used by engineering teams via various devices, such as computers, tablets, and mobile phones, but also on large displays placed throughout the factories floors. MyWorkplace offers users the flexibility to construct and customize their own dashboards, improving their experience with the application itself as well as with the integrated widgets. Each user

can customize their own dashboards to match their requirements and share them with others or with their own team. The application provides access to a selection of integrated applications within the portal and pre-set dashboards. Furthermore, the MyWorkplace's extensibility allows external teams to effortlessly create and integrate their own widgets onto the platform. The MyWorkplace team is composed of three internal teams at CTW alongside one external team. MyWorkplace manages a set of five distinct applications known as *Core Apps*, which have been integrated into the portal using iFrames. These teams have developed and currently maintain ten widgets and eleven web components with one team dedicated primarily to widget development while the remaining teams focus on other projects. These widgets and web components, alongside the Core Apps share a codebase inside a monorepo created with Nx Workspace (described in detail in section 2.2.4) enabling shared libraries to be used across all existing applications. The external team is responsible for two applications integrated into the MyWorkplace portal, as well as four widgets which they develop and maintain. The MyWorkplace relies on an extensive stack of technologies. On the frontend, this includes AngularJS, Nx Workspace, Angular, and the company's Design System. However, AngularJS technology was deprecated in 2022, leading to the need to migrate the application to a new technology (Thompson, 2023). In order to achieve this migration in a less impactful way, web components are now being developed using Angular. Both the web components used in the migration from AngularJS to Angular are developed by integrating these Angular micro applications into an Nx Workspace.

1.2 Problem Description

The MyWorkplace application is at a critical point where its growth and complexity have started to impact its performance, particularly during startup. This challenge comes

from the application's extensive need of resources, including images, fonts, dynamic and Javascript bundles from its integrated widgets, applications, and web components. However, the cumulative weight of these resources has led to prolonged loading times. Users, expect applications to be responsive and quick, especially for users that are located in physical places far away from the server physical locations. Any delay, especially during startup affects user overall satisfaction. Another layer to this challenge is the structure and size of the final bundles. These bundles, which are meant to package all shared essential dependencies, have grown considerably in size. These dependencies serve multiple applications and it plays a big collective impact on response times from the application. Figure 1.1 represents the requests and response times being made by the application prior to any implementation of Module Federation.

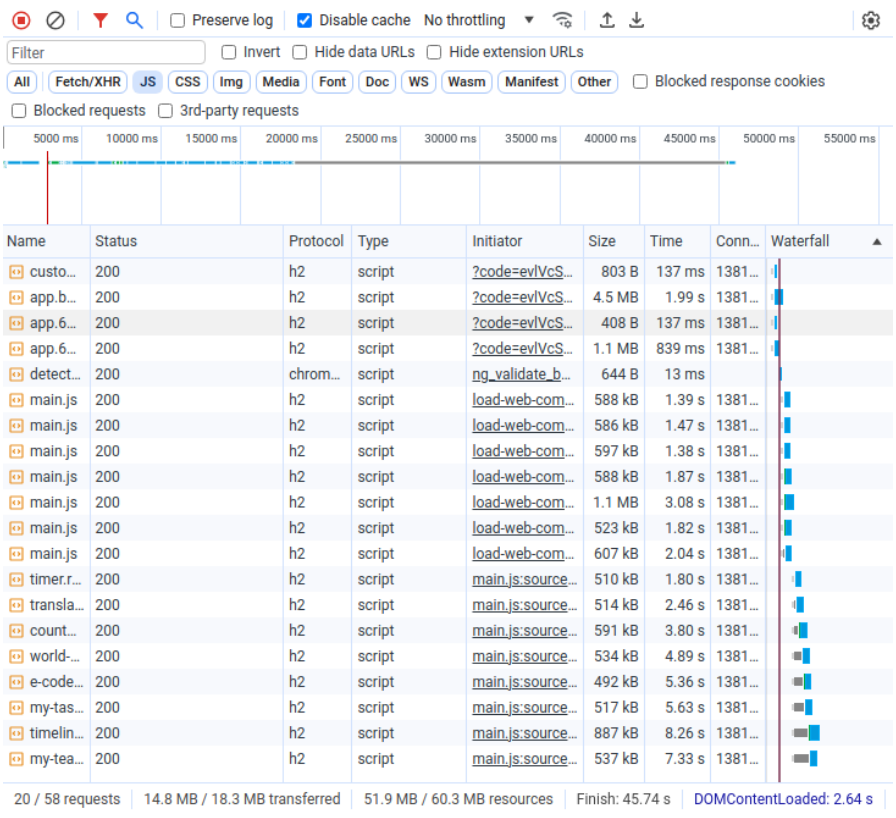


Figure 1.1: Network print screen before adding Module Federation.

In essence, the core problem can be resumed to the need to optimize the resources in order to ensure a faster application start and the challenge of managing a growing library of dependencies without compromising on performance. Addressing these issues is crucial for the future scalability and performance of the application.

1.3 Objective

The goal is to compare the solutions available in the market, identify and create a solution that meets the product's needs. This solution aims to optimize and accelerate the MyWorkplace application, while preserving the ability to support new developments and the addition of new features.

The speed and reliability of the product are of utmost importance, as its performance can have a significant impact on the production line. Therefore, it is crucial that these aspects are taken into consideration when choosing the solution.

1.4 Methodology

The approach taken to address the problem outlined in this document follows the best practices and concepts taught during the academic journey, particularly regarding the process of analyzing, designing, and implementing the solution.

The methodology used was the same as the one used by the internal teams at CTW, who follow the LeSS (Large-Scale Scrum) framework, which allows them to have ownership over the design, planning, and execution of tasks, progress, and associated work processes. (LeSS, 2023).

A monthly macro plan was created in order to guide the development of the thesis within the scope of the team and project. It is worth noting that these stages can be adjusted

according to the work flow considering the progress of practical work and thesis writing. The outcome of the planning by monthly intervals is presented in table 1.1.

Table 1.1: Work Plan

Tasks	Months Interval
• Analysis and understanding of the current problem with the MyWorkplace application, as well as its requirements and objectives. • Research of existing solutions in the market for optimizing resource loading in web applications. • Development of a detailed work plan: establish objectives, stages, and deadlines for the rest of the thesis.	[1-4]
• Start of thesis writing, including the description of the problem, research of existing solutions, implementation and optimization of the chosen solution, as well as test results	[1-9]
• Implementation of the chosen solution, in order to optimize resource loading and minimize the size of the final bundles. • Conduct tests to verify the effectiveness of the implemented solution, including measuring loading times and the size of the final bundles.	[4-7]
• Continuous optimization of the implemented solution, including identifying and correcting issues. • Additional testing: conducting additional tests to verify the effectiveness of the optimized solution.	[7-9]
• Review and conclusion of the thesis: review and conclude the thesis, including checking for consistency and coherence in the text, as well as performing final revision to fix any errors.	[10-end]

1.5 Contributions

The main contributions of this project are:

- Systematization of concepts, actors and business processes;
- Analysis of existing solutions in the market, identifying the advantages and disadvantages of each one;
- Development of a solution that leads to an improvement in the loading speed and overall performance of the MyWorkplace application.

1.6 Thesis Structure

In the first chapter, the topic of this dissertation is introduced, as well as the problem it aims to solve.

In the second chapter, the context is systematically detailed, including a description of the product's technology stack.

In the third chapter, its introduced a overview and historic context of modularity and existing solutions in the market are presented and compared to the technologies that could be adopted to achieve the objectives in question.

In the fourth chapter, an extensive analysis of Module Federation is conducted. It is explained its features and applicability as well as some case studies are presented and discussed.

In the fifth chapter, the system design is presented using different levels of granularity.

In the sixth chapter, concepts related to the implementation of the system are discussed, including technological decisions, tools used.

In the seventh chapter, is presented analysis and an evaluation of the implemented solution.

1.7 Summary

This provides context, explains the problems faced in the current setup within the project, and outlines the objectives this document aims to achieve. It emphasizes the contributions of the thesis and concludes with an overview of the thesis structure, ensuring a logical flow for detailed exploration and analysis in the subsequent chapters.

Chapter 2

Context

This chapter aims to present a description of the technological context, identifying existing applications in the product as well as the technological stack present in the front-end.

2.1 Existing Applications

In the upcoming sections, the existing application types in the product, as well as the type of integration in the MyWorkplace portal, will be presented. The Figure 2.1 illustrates this different types of applications present in the MyWorkplace context, marked with different colors in order to distinguish one from another.

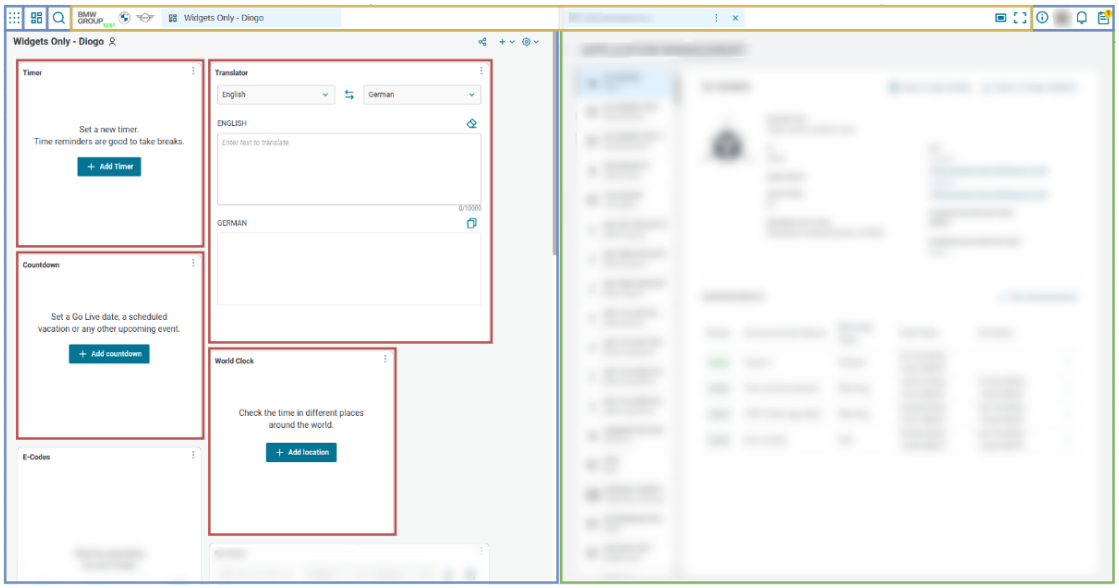


Figure 2.1: MyWorkplace application context.

The yellow rectangles represent the MyWorkplace Shell, the red ones the widgets, the blue ones indicate the Angular-Web-Components, and the green ones represent both the External Apps and Core Apps.

2.1.1 MyWorkplace Shell

The MyWorkplace Shell application, developed using AngularJS, serves as a single entrypoint to access both Core Apps and Widgets. This allows users to access all the information of multiple applications in one place. The Core Apps can be developed by both MyWorkplace teams and external teams, and their integration within MyWorkplace is achieved through the use of iFrames. The Widgets are integrated in the form of web components, allowing for their reuse throughout the application. The Web-Components, are as the name indicates, integrated in the application by the means of web components. Along with the application integrations, MyWorkplace Shell enables users to create personalized and shared dashboards, providing an intuitive way to interact with various

applications and widgets.

2.1.2 Core Apps

The Core Apps are self-contained applications built using different versions of Angular. Each of these applications has its own domain and can be used independently. There are five different Core Apps, all of which are integrated into the MyWorkplace Shell. This applications are developed and maintained exclusively by the MyWorkplace teams. A couple of examples of Core Apps are:

- App Management
- User Settings

2.1.3 External Applications

Applications developed and maintained by teams outside of MyWorkplace are referred to as External Applications. These applications are integrated into the portal through the use of iFrames and can be developed using various frameworks and libraries, such as Angular, React, VueJs among others.

2.1.4 Angular-Web-Components

In the context of this thesis and in order to reduce confusion and differentiate the domain "web-components" from the application type "web components", this domain objects will be referenced as Angular-Web-Components and the type of applications will still be called web components. Web Components are a group of technologies that allows to create reusable encapsulated custom elements. These elements can be integrated into web applications without interfering with other parts of the code (Mozilla, 2023a). The core of Web Components consists of three main technologies:

- Custom Elements: JavaScript APIs that define and manage custom elements;
- Shadow DOM: Ensures encapsulation by attaching a separate DOM tree to elements, keeping them isolated;
- HTML Templates: Using elements like and , can be defined markup structures that aren't immediately rendered, allowing for flexible content insertion (Mozilla, 2023a).

Angular-Web-components are small applications being built to facilitate the migration of MyWorkplace Shell from AngularJS to Angular. These components are integrated through web components, making them reusable within the MyWorkplace Shell. Migrating to a new version of Angular using web components enables the addition of new features on top of legacy code, maintaining it while providing users with new functionalities. Some examples of Angular-Web-Components are:

- Apps Menu
- Dashboards
- Dashboards Menu

2.1.5 Widgets

Widgets are a collection of small applications dynamically integrated into different dashboards in the MyWorkplace Shell. These applications, like the Core Apps, originate from both external teams and MyWorkplace teams. The Widgets developed by the MyWorkplace teams are all built using the Angular framework, while other widgets developed by external teams are built using different technologies such as React or VueJs. In order to make it easier to dynamic integrate widgets into dashboards, these Widgets are built using web components.

These applications are compiled in a way that makes everything accessible from the main

entry point in a single bundle, meaning that both scripts and styles remain in their own bundle, allowing the application to make a single request for each type of widget, thus reducing the number of requests made by the MyWorkplace Shell. Some examples of Widgets are:

- Timer Widget
- Countdown Widget
- Timeline Widget

2.2 Technological Stack

In the following sections, it will be presented the technologies used by the applications that make up the MyWorkplace product.

2.2.1 Angular

Angular is a development platform built to meet the demands of building scalable web applications. This development platform is built on TypeScript and it is a component-based framework that comes with a rich collection of integrated libraries, as well as a set of tools and features that enable developers to build modern and efficient applications (Angular, 2022).

2.2.2 Webpack

Webpack is a static module bundler for JavaScript applications, allowing developers to work with multiple files and dependencies, and bundling them into one or more output files. This technology can handle JavaScript modules and other types of resources, such as image and font files, using a dependency graph to analyze and compile modules. Webpack

supports the use of plugins and loaders, allowing to extend its functionalities and work with additional file types (*Concept*, 2023). One of its main advantages is the ability to split the code into multiple output files, significantly improving the application's loading time and reducing the size of the final file. This allows the application to load only the necessary resources at a given moment, increasing the speed and efficiency of the system (*Concept*, 2023; Koppers, 2023).

2.2.3 AngularJS

AngularJS is an open-source JavaScript framework for web application development and has been a leading framework in the world of JavaScript frameworks. Hundreds of websites and applications have been built on top of this framework, however, this technology has been deprecated starting from 2022 (Maida, 2017; Thompson, 2023).

2.2.4 Nx Workspace

Nx Workspace is a project management tool in monorepo format that aims to increase the development process with minimal effort. One of the main advantages of using Nx Workspace is distributed computation caching feature, which allows sharing the cache of build results from applications and libraries among Continuous Integration/Continuous Delivery (CI/CD) machines and local machines. Additionally, this tool also offers a large number of plugins for generating, compiling, and testing different frameworks (Nrwl, 2023).

2.3 Interconnection of Applications

The Figure 2.2 represents the relationships between the different types of applications mentioned in section 2.1.

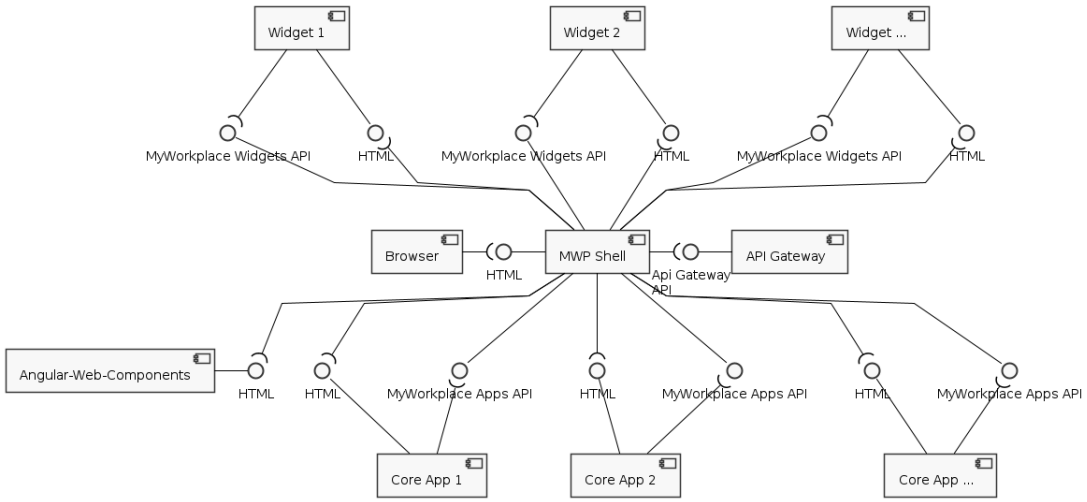


Figure 2.2: Component arquitetural representation of MyWorkplace products.

Widgets communicate with the MyWorkplace Shell through an API that exposes certain functions, enabling Widgets to perform specific actions. Similarly, the Core Apps also communicate with the MyWorkplace Shell through their own API. Both of these APIs are optional and not required to be used by either widgets or Core Apps. The MyWorkplace Shell makes requests to a GraphQL API Gateway, while standalone applications inserted into the portal make requests to different APIs, which are not represented in the diagram in order to simplify its interpretation.

2.4 Summary

In this chapter, a brief overview of the product applications and their integration within the MyWorkplace portal was presented. Additionally, the technological context that motivated the creation of this project was introduced.

Chapter 3

Modularity in Javascript

The purpose of this chapter is to introduce the concept of modularity in software development. This chapter offers insights into the evolution of modularity over time in the context of Javascript. The Modularity chapter discusses and presents the current state of modularity in Javascript, presenting a comparison between static and dynamic modules and also describes the tools that aid in the management and bundling of modules.

3.1 Definition

A module is a cognitive or perceptual subsystem whose functioning is relatively independent. This module should be independent of the rest of the cognitive architecture and its functioning can be analyzed and understood relatively independently of the overall system in which it is embedded. A module is a component whose behavior is not influenced by any other process or activity that occurs in the system in which it is integrated, beyond what is provided to these components as input (McClamrock, 2006).

3.2 Historical Context

Initially, the JavaScript language was used embedded in Hypertext Markup Language (HTML) through `<script>` tags or through JavaScript files, which shared the global scope of the page. As a result, any of the files could write variables to the global `window` variable, which could easily lead to conflicts and inadvertently cause one script to overwrite a variable of another script (Bevacqua, 2020; Farrell, 2019). Figure 3.1 illustrates this example of global scope sharing.

```
1    <script>
2        var loading = true
3    </script>
4
5    <script>
6        if(loading){
7            console.log('Has loaded.')
8        }
9    </script>
```

Listing 3.1: Use of script tags.

3.2.1 Immediately-Invoked Function Expression

The problem of global scope pollution led to the creation of Immediately-Invoked function expressions (IIFE). IIFE define an anonymous function inside a group operator, creating a new level of scope, preventing access to the `var` variables defined inside the IIFE, while minimizing pollution of the global scope (Bevacqua, 2020; Mozilla, 2023b). Figure 3.2 illustrates an example of using IIFE notation, where the variables `firstVariable` and

`secondVariable` will be discarded after the function is executed. This pattern allowed multiple IIFEs to be concatenated into a single file, reducing stress on the network by reducing the number of requests. However, this pattern does not allow the creation of a dependency tree, requiring files to be ordered precisely to avoid dependencies loading before any module that depends on them, leading to recursion problems (Bevacqua, 2020).

```
1    (() => {
2        // some initiation code
3        let firstVariable;
4        let secondVariable;
5    })();
```

Listing 3.2: Use of IIFE

3.2.2 RequireJS

In order to address the issues raised in section 3.2.1, some libraries emerged, such as the example of RequireJs library or the dependency injection mechanism in AngularJS, but in neither case were these libraries adopted as a specification (Bevacqua, 2020; Farrell, 2019). Figure 3.3 illustrates an example of using the RequireJS library and how, unlike IIFE, it explicitly declares dependencies at the module level, making the relationships between the component and other parts of the application obvious. Another advantage of using this library compared to the IIFE functions is that RequireJS resolves the dependency tree regardless of the number of modules (Bevacqua, 2020).

```
1    define(function (require) {
2        // Loading app-specific modules
3        var messages = require('./messages');
4        // Loading library/vendor modules
5        var print = require('print');
6        print(messages.getHello());
7    });
```

Listing 3.3: RequireJS usage.

3.2.3 CommonJS

Node.js introduced the CommonJS (CJS) modules as its original way of packaging JavaScript code. These modules are loaded synchronously and allow reducing boilerplate code when compared to RequireJS or AngularJS. Unlike these solutions, CJS modules are strict, meaning that each file can only have one module and there is no possibility of having multiple modules dynamically defined per file. Although not an official specification, CJS modules are widely used (Bevacqua, 2020; Mohan & Prusty, 2018; Node.js, 2023). Figure 3.4 illustrates the import and usage of a CJS module.

```
1    const circle = require('./circle.js');
2    function area(){
3        console.log(`Radius: 4; Area: ${circle.area(4)}`);
4    }
5    // exposes area function to other modules
6    exports.area = area;
```

Listing 3.4: Use of CommonJS.

The adoption of this type of module made it easier for tools to understand the hierarchy of the CJS component system. Since each file is considered a module, using the `require` function would load its dependencies, and any assignments to module.exports defined its interface. In order to connect these types of modules to the browser, browserify was created, which allowed the `require` method, previously exclusive to Node.js, to be used in browsers (Bevacqua, 2020; browserify, 2023).

3.2.4 ECMAScript Modules

In June 2015, the ECMAScript 6 (ES6) specification became a standard, which included an official native syntax for JavaScript called ECMAScript Modules (ESM). These modules create a dependency graph where the connections between nodes are the imports and the nodes are the JavaScript files (Bevacqua, 2020; Clark, 2018).

```
1  import { circle } from 'circle';
2
3  // exposes area function to other modules
4  export function area(){
5      console.log(`Radius: 4; Circle Area: ${circle.area(4)
           }`);
6  }
```

Listing 3.5: Usage of ESM

The instances of ESM modules are created in three stages:

- **Compilation** - This phase involves the search, download (either through a Uniform Resource Locator (URL) or through the file system), and parsing of all files into data structures that browsers can use, called Module Records.

- **Instantiation** - This phase involves the allocation of memory for the exported values. Subsequently, through a process called linking, the imports and exports are pointed to these memory locations.
- **Evaluation** - This last phase executes the code and fills the location in memory with the values.

These stages can run separately, which means that ESM modules can be considered asynchronous (Clark, 2018). This is considered an advantage over CommonJS, given that this functionality means that certain parts of the application's dependency graph can be loaded concurrently or lazily in response to specific events. Another advantage over CommonJS is the use of static imports. These static imports can be statically analyzed and lexically extracted from the Abstract Syntax Tree (AST) of each module in the system, leading to an improvement in the introspection capabilities of module systems (Bevacqua, 2020).

3.3 Static vs Dynamic Modules

Rauschmayer (2022) defines the terms "static" and "dynamic" as adjectives that describe different phenomena in programming languages:

- Static - Something related to the source code that can be determined at compile-time;
- Dynamic - Something that is determined at runtime.

3.3.1 Static Modules

Static modules are modules that are resolved at compile-time and imported using a static import declaration. These declarations can only be present at the top level and only

exist in modules. They are syntactically rigid, allowing these modules to be statically analyzed and linked before being evaluated. There are four different types of static import declarations:

Named Import

This type of static import allows importing multiple names from the same module. These values must be exported by the module. Code snippet 3.6 represents an example of a named import.

```
import { export1 , export2 } from "module-name";
```

Listing 3.6: Named imports usage example. Retrieved from MDN contributors, 2023b.

Default Import

Allows importing modules that have been exported in the form of a default export. Example of a default import is shown in code snippet 3.7.

```
import defaultExport from "module-name";
```

Listing 3.7: Usage of Default imports. Retrieved from MDN contributors, 2023b.

Namespace Import

The namespace import statement allows the insertion of a namespace object containing all the exports from the referenced module. An example of a namespace import is shown in code snippet 3.8.

```
import * as name from "module-name";
```

Listing 3.8: Example of usage of Namespace imports. Retrieved from MDN contributors, 2023b.

Side Effect Import

Enables importing an entire module only for its side effects. This import runs the code in the global scope without importing any values and is often used for modules that change global variables, such as polyfills. Code snippet 3.9 represents an example of a Side Effect Import.

```
import "module-name";
```

Listing 3.9: Side Effect Imports usage. Retrieved from MDN contributors, 2023b.

To be able to incorporate these static modules, it is necessary for the file to be dynamically interpreted as a module during runtime. To achieve this, the type="module" is added to the `<script>` in HTML.

3.3.2 Dynamic Modules

Recent advancements in JavaScript have enabled the dynamic loading of JavaScript modules, providing the ability to load modules only when required, as opposed to loading all modules initially, as is the case with static modules. This functionality leverages the use of the import() function, which returns a Promise that resolves to a JavaScript module, allowing access to its exports. Listing 3.10 provides an example of dynamically importing a module (MDN contributors, 2023b).

```
1  import("./modules/myModule.js").then((module) => {
2  // Do something with the module.
3  });
```

Listing 3.10: Example of usage of dynamic imports. Retrieved from MDN contributors, 2023b.

In JavaScript, this dynamic import is an expression that enables the asynchronous and dynamic loading of ECMAScript modules in both module environments and other environments, which, in turn, enables the use of this function even within `<script>` tags that are not of the module type (MDN contributors, 2023b, 2023a).

3.3.3 Comparison

Table 3.1 summarizes the analysis of the two types of module loading according to their characteristics.

Table 3.1: Comparison of dynamic and static module loading.

	Static Import	Dynamic Import
Application Loading Speed	Slower	Faster
Memory Usage	Consumes More Memory	Consumes Less Memory
Dynamically Constructed Import String	No	Yes
Conditional Import	No	Yes
Environment	Module Type	All Environments
Tree-Shaking	Supports	Does Not Support
Static Analysis Tools	Supports	Does Not Support

Static module imports result in module evaluation at compile time, rather than dynamic modules that are evaluated at runtime. Static module imports should be used to load initial dependencies and allow bundlers to utilize tree shaking. The use of modules through static imports makes it easier to take advantage of static analysis tools. However, static imports also significantly slow down loading, and these initially loaded modules may not be needed during the application's use (MDN contributors, 2023b, 2023a). Dynamic imports allow loading modules that don't exist at compile time, enable the specification of dynamically constructed strings for module import, and allow conditional module importing. Conditional module import can be crucial for importing modules that have side effects, with these side effects being desirable under certain conditions (MDN contributors, 2023b, 2023a).

3.4 Module Bundler

A Module Bundler is a tool that facilitates the software compilation process. It resolves code dependencies and can typically remove unnecessary dependencies in production environments, a process known as tree shaking (Latendresse, Mujahid, Costa, & Shihab, 2022). Figure 3.1 represents a trend graph from 2021-2022 showcasing the usage of the bundlers described in this section (NPM Trends, 2023).

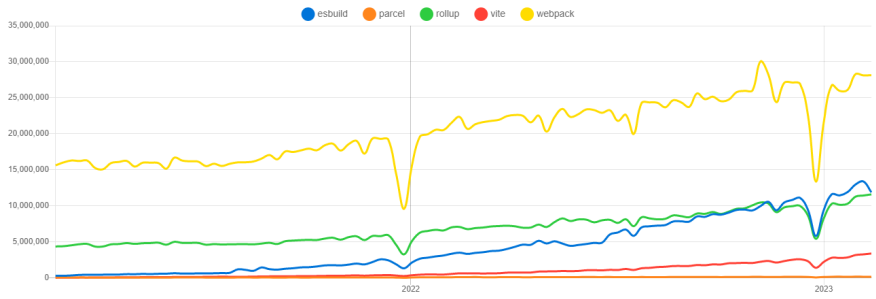


Figure 3.1: Trend graph from 2021-2022 showing the usage of different module bundlers, adapted from NPM Trends, 2023

3.4.1 Webpack

Webpack is a bundler that, by defining an entry point in the application, analyzes all dependencies and generates JavaScript bundles that include all the necessary code for the application to run. Webpack supports the use of plugins, which can be used to optimize code, remove unused code parts, minify code, among other functionalities (*Concept*, 2023).

3.4.2 esBuild

esBuild is a JavaScript bundler created by Evan Wallace. It is identified as being up to ten times faster than its alternatives. This bundler is written in the Go programming language and compiles to native code. It supports plugin creation and integration and offers features such as minification, tree shaking, source maps, watch mode, and a local server, among others (Wallace, 2023).

3.4.3 Rollup

Rollup is a JavaScript module bundler that focuses on ES6 modules. It offers the possibility to integrate with other tools, such as Deno and Babel, and allows plugin creation and integration. Rollup provides capabilities for tree shaking and code splitting (RollupJs, 2023).

3.4.4 Parcel

Parcel version 2, created in 2021, is written in Rust and is based on the Speedy Web Compiler (SWC). Parcel performs builds in parallel using worker threads, allowing it to utilize all available machine cores. This bundler utilizes its own cache, eliminating the need to compile the same code more than once (Parcel, 2021, 2023).

3.4.5 Comparison

Wallace (2023), the creator of the esBuild bundler, conducted benchmark tests comparing different bundlers by analyzing a JavaScript project that includes ten duplications of the `three.js`[1] library. These tests show that esBuild is faster than the alternatives, with Webpack being the slowest of all the analyzed bundlers. The results of these tests are represented in Figure 3.2.

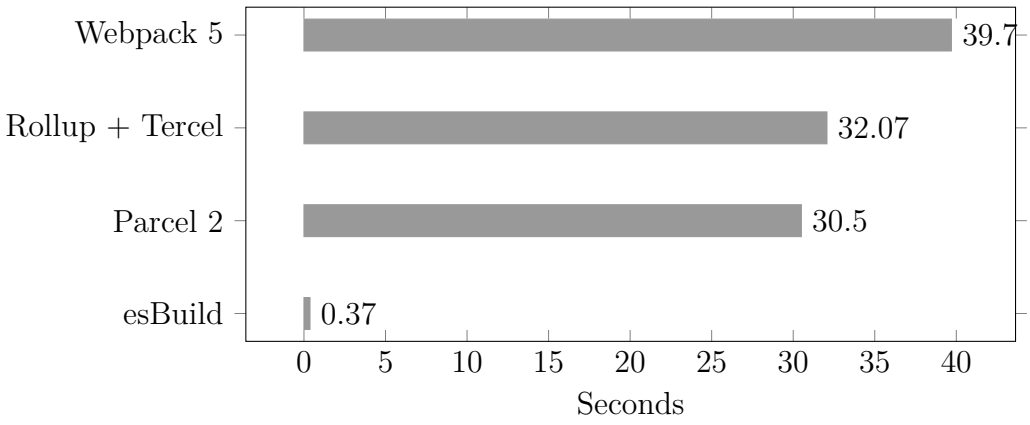


Figure 3.2: Comparative benchmark of different bundlers, adapted from Wallace, 2023

Tests were also conducted for projects using TypeScript, but the values were similar to the test conducted on a JavaScript project, where esBuild was the fastest and Webpack 5 was the slowest. Taking into account these values and other properties of each bundler, Table 3.2 presents a comparison of the different module bundlers described in Section 3.4.

[1]This analysis may be considered partial, as it was conducted by the creator of one of the analyzed bundlers. It is worth noting that the repository where this analysis was performed is public domain, and this evaluation can be repeated considering the source code present in the repository (https://github.com/evanw/esbuild).

Table 3.2: Comparison of properties of different module bundlers.

	Webpack	Rollup	Parcel	esBuild
ES Modules Support	Yes	Yes	Yes	Yes
TypeScript Support	Yes[a]	Yes[a]	Yes	Yes
Code Splitting	Yes	Yes	Yes	Yes
Popularity[b]	62.6k	23k	42k	34.7k
Tree Shaking	Yes	Yes	Yes	Yes
Hot Module Replacement	Yes	Yes	Yes	Yes
Performance[c]	Low	Medium	Medium	High

[a] requires additional configuration, [b] evaluated based on the number of stars on GitHub as of 02-26-2023, [c] performance evaluated based on the benchmark values analyzed.

According to the analysis, module bundlers vary primarily in terms of popularity and performance. Among them, Webpack stands out as one of the most renowned. While not highlighted in this particular comparison, it's noteworthy to mention a few emerging module bundlers such as Rspack, Turbopack, and Bun Bundler, which promise enhanced performance compared to their more established counterparts.

3.5 Summary

This chapter defined and described the concept of modularity and how it has evolved and exists in the context of JavaScript. It also presented different JavaScript module bundlers and their specific characteristics.

Chapter 4

Module Federation

This chapter discusses Module Federation, exploring its core concepts, operations, and practical applications. It offers insights into its functionality, case studies, and alternatives. This exploration aids in understanding and utilizing Module Federation efficiently in development environments. It is important to note that the references in this chapter will not be predominantly scientific. Given the recency of Module Federation, much of the insights and information are retrieved from the practical field. The references will predominantly include conferences, examples, videos, and blog posts from the author of Module Federation that explains this topics and also recreates complex and practical examples of case studies using Module Federation.

4.1 Definition

Module Federation was created by Zack Jackson, with co-authors Marais Rossouw and Tobias Koppers. It is an integral part of Webpack 5, which was made available in October 2020. It introduces a new way to import chunks, features, or dependencies from another independently deployed application into an application. The remotely imported code can be used as an ES6 import, as if it were part of the same repository, without being tied to any specific framework (Ebey, n.d.; Ghadyani, 2021; Silva, 2021). Figure 4.1 illustrates an example of three applications (sites in the diagram) sharing modules at runtime.

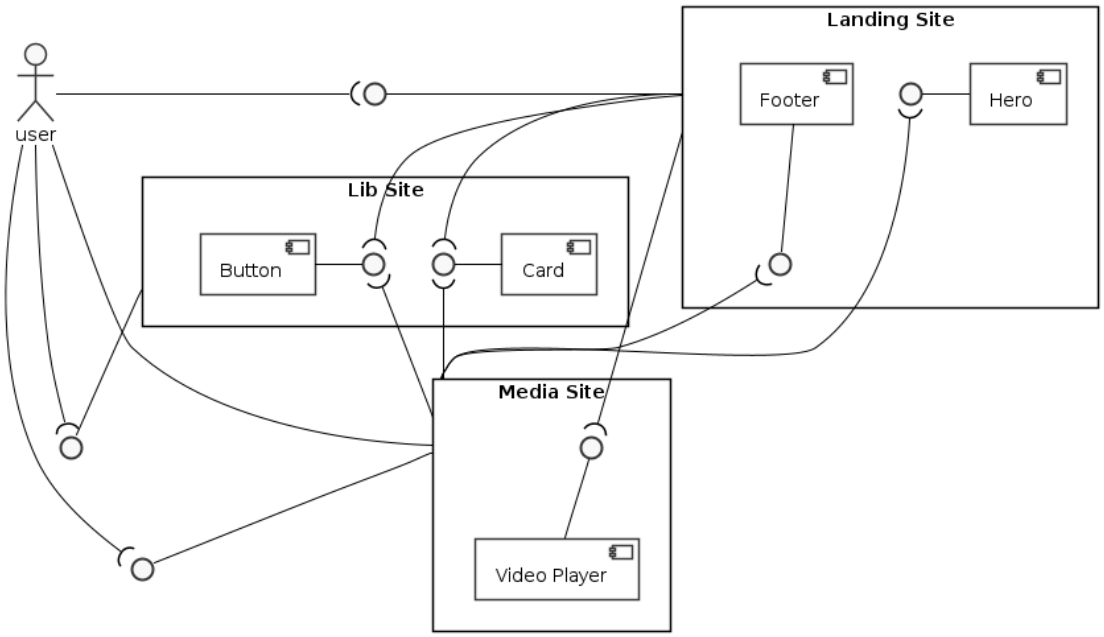


Figure 4.1: Three different applications share modules at execution time. Adapted from Ebey, n.d

Module Federation introduces the following terminology, as defined by its creator, Zack Jackson (2023b):

- *Host* - A Webpack build that is initialized first during page loading;
- *Remote* - A Webpack build where a part of it is consumed by a host application;
- *Bidirectional-hosts* - A Webpack build that can function both as a host, consuming remotes, and as a remote. These two roles are not mutually exclusive, and an application can be both a host and a remote;
- *Omnidirectional-hosts* - At application startup, it is not known whether it is a host or a remote. This allows Webpack to dynamically negotiate and switch vendors using semantic versioning, allowing different versions to coexist.

The configuration of a remote application using the `ModuleFederationPlugin` is done in a Webpack configuration file. An example of this configuration is shown in code snippet 4.1.

```
1     output: {
2         publicPath: "http://localhost:5000/",
3     },
4     plugins: [
5         new ModuleFederationPlugin({
6             name: "appRemote",
7             filename: "remoteEntry.js",
8             exposes: {
9                 './component': "./host/component"
10            },
11            shared: ["rxjs"]
12        }),
13    ],
```

Listing 4.1: Example of a remote configuration in a *webpack.config.js* file.

The remote application exposes the `./host/component` module to be consumed by the host application. It also expects the rxjs library to be imported by the host application without including it in the application bundle. The configuration of a host application using the `ModuleFederationPlugin` is similarly done in a Webpack configuration file. An example of this configuration is shown in code snippet 4.2.

```
1       plugins: [
2           new ModuleFederationPlugin({
3               name: "app2",
4               remotes: {
5                   appRemote: "appRemote@http://localhost:5000"
6               },
7               shared: ["rxjs"]
8           })
9       ]
```

Listing 4.2: Example of a host configuration in a *webpack.config.js* file.

The code snippets configurations in 4.1 and 4.2 use three fundamental concepts:

- *Remotes* - A set of names from other applications using Module Federation that this application will consume. In Figure 4.2, the application `app2` consumes code from the application named `appRemote`.
- *Exposes* - Represents the set of files the application exports as remotes to other applications. In Figure 4.1, the application `app2` exposes the module `./host/component` as `./component` to be consumed by its host application.

- *Shared* - A list of libraries that the application shares with other applications.

The Module Federation concept is agnostic to the environment. This means it can be applied in web platforms, Node.js environments, or others (Webpack, 2023a).

The importance of using Module Federation in MyWorkplace becomes evident when considering the challenges faced in previous contexts. Prior to its adoption, the team relied on Webpack externals to minimize library code duplication in the widgets. This approach used the Universal Module Definition (UMD) to register these libraries within the window object. However, certain libraries stopped supporting UMD, and the team's commitment to maintaining the most up-to-date application possible made the use of externals impossible. Consequently, widgets began to exhibit a larger footprint, characterized by substantial bundles. This led to the need for a solution like Module Federation, which promised to address these issues.

4.2 Compilation Time

Module Federation is exported by the Webpack bundler in the form of a plugin called `ModuleFederationPlugin`. This plugin abstracts the implementation of two other plugins (which can be used independently depending on whether the application is a `remote` or a `host`):

- **ContainerPlugin** - This plugin is responsible for creating a new entry, which serves as an application's manifest, where the modules to be exposed are specified (Webpack, 2023a). This is the plugin that the remote application uses.

- **ContainerReferencePlugin** - This plugin manages the `remotes` modules in the configuration, adds references to containers as externals, allowing the import of remote modules from these containers (Webpack, 2023a). This is the plugin that the `host` application uses.

Compiling a `remote` application creates JavaScript files, not only to run the application itself but it also creates the JavaScript bundles for the remote modules (Webpack, 2023a).

4.3 Runtime

At runtime, the browser loads the remote entry file. This process registers a global variable with the name specified in the library parameter of the ModuleFederationPlugin configuration. This global variable exports two key functions: `get` and `override`. The `get` retrieves remote modules, while `override` manages all the shared libraries (Herrington & Jackson, 2023). When a `remote` module is exported, the module's implementation can be accessed via the `window` object, exemplified in code snippet 4.3.

```
1  window.appRemote.get('component')
2    .then(factory => console.log(factory()));
```

Listing 4.3: Example of accessing modules from remote applications.

The invoked `factory()` returns an object. For `named exports`, these names are mapped to this object. This factory enables Webpack to load both the module and its required `shared` dependencies. These dependencies are loaded only if they

haven't been loaded by another remote or host. Module Federation resolves the dependencies not only of the direct modules but also the `remotes` modules, if these modules are bidirectional-hosts. Circular dependencies are allowed and are efficiently managed by Module Federation (Herrington & Jackson, 2023).

4.4 Versions

In the expansive Javascript ecosystem, the Node Package Manager (NPM) recommends adhering to the Semantic Versioning specification. To do so, it's necessary to update and publish an updated version of the package in package.json. This versioning system helps teams that depend on a certain package understand how a change in a particular version can affect their code and allows them to make the necessary adjustments. Semantic Versioning delineates package versions using three distinct numbers, separated by a dot (".") (npm, 2021; Preston-Werner, n.d.). When working with Module Federation, version specifications align with Semantic Versioning. Teams with applications that consume external modules can encounter three problems:

- Duplication - This is common when libraries and packages are not shared. This can lead to a poor user experience of the applications (Herrington & Jackson, 2023).
- Version Mismatch - This problem occurs when an application consumes packages in versions that are not compatible with the current version used in the application. In extreme cases, this mismatch can cause the application not to run (Herrington & Jackson, 2023).

- Singletons and Internal State - Some libraries have an internal state, and this internal state is required at runtime for the library to run without problems. For these libraries, there can only be one instance of the library at runtime (Herrington & Jackson, 2023). Module Federation introduces measures to solve or diminish these issues. The `ModuleFederationPlugin` provides the possibility of adding a `shared` key to its configuration. This attribute allows declaring libraries that will be shared between applications, eliminating duplication of these modules. This configuration is depicted in Figure 4.2 (Herrington & Jackson, 2023).

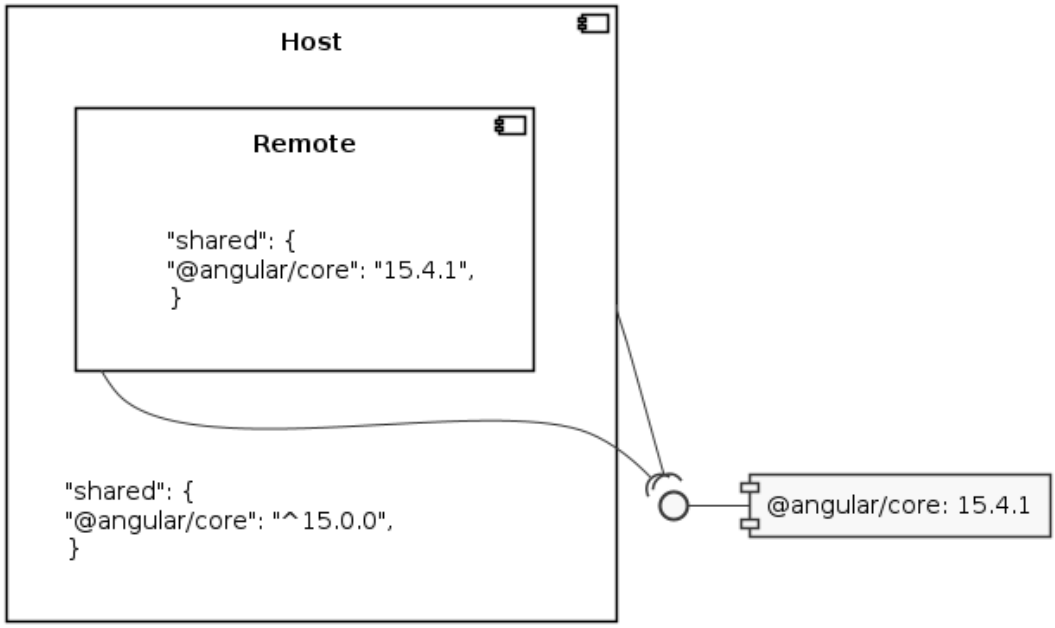


Figure 4.2: Representation of version sharing.

Webpack solves version mismatches using fallbacks for shared libraries (Herrington & Jackson, 2023). As illustrated in Figure 4.3, both host and remote applications consume the `@angular/core` library. Even though both applications include this

version under the `shared` attribute, they depend on different versions. The host application first loads the version of `@angular/core` it needs, then the remote application starts loading and searches for its version, but due to a version mismatch, the library's fallback is used.

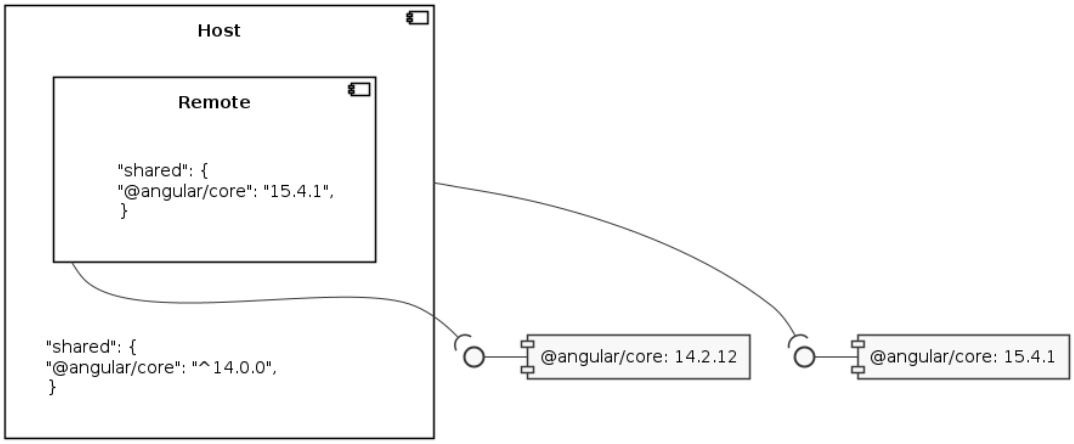


Figure 4.3: Representation of version mismatch handling.

4.5 Applicability

In this section, the potential uses of Module Federation in various scenarios will be described, whether by migrating existing applications, adding new functionalities, or migrating architectures.

4.5.1 Micro-Frontends

Micro-Frontends is an approach to decomposing the frontend into individual and semi-independent micro-applications (Taibi & Mezzalira, 2022). This approach mirrors the microservices architecture, where backend development splits into logically independent units. Micro-frontends architecture allows the division of

a monolithic application into several parts that can be developed simultaneously by different teams. This architecture has been adopted by various companies, including IKEA, DAZN, Starbucks, among others (Mezzalira, 2019; Nishizu & Kamina, 2022; Taibi & Mezzalira, 2022).

Taibi & Mezzalira (2022) proposed a decision framework using four key decisions to start a Micro-Frontends project: Horizontal or Vertical split, Composition Side, Routing, and Micro-Frontends communication.

Horizontal or Vertical split

The decision aims to identify whether multiple frontends are wanted in the same view (horizontal split) or if a view or group of views is intended to be assigned to a team (vertical split). Figure 4.4 illustrates the difference between horizontal split and vertical split.

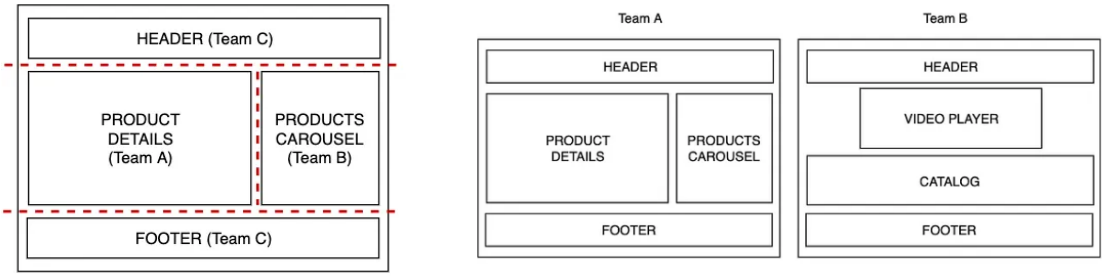


Figure 4.4: *Horizontal split* (esquerda) vs *Vertical split* (direita). Retirado de Mezzalira, 2019

Composition Side

The decision responsible for identifying the type of composition to use in the architecture, client-side composition, server-side composition, or edge-side composition (Taibi & Mezzalira, 2022). Figure 4.5 graphically illustrates the different types of

Micro-Frontends composition.

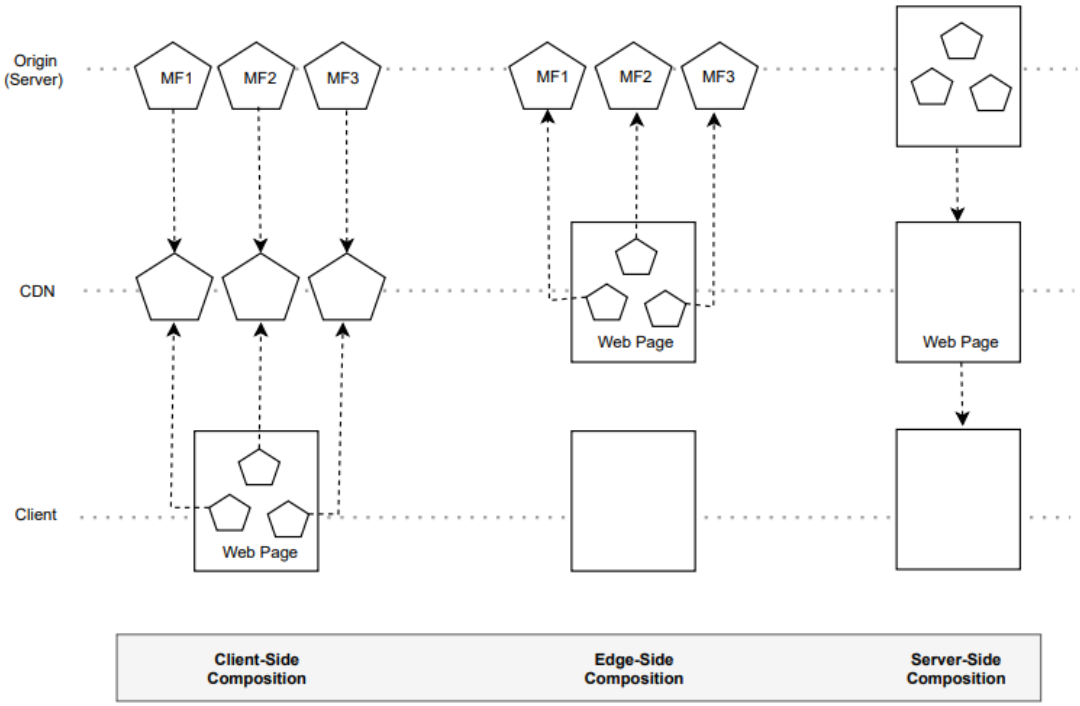


Figure 4.5: Comparison between different types of Micro-Frontends. Retrieved from Taibi & Mezzalira, 2022

- *Client-side Composition* - Allows combining and integrating micro frontends in the browser.
- *Server-side composition* - Involves having a backend service responsible for combining pages using pre-rendered sources.
- *Edge-side composition* - Uses a markup language called Edge-side Includes (ESI) proposed by Akamai to combine pages using various Content Delivery Networks (CDNs) or proxies responsible for storing cache and delivering content close to the region of the world where the request was made (Silva, 2021; Taibi & Mezzalira, 2022).

Routing

A decision that consists of choosing the best way to create routes from view to view.

Micro-Frontends communication

A decision that aims to identify how micro frontends will communicate with each other, either by the usage of web storage, cookies, data in the format of query strings, through Web Component events, or any other communication method.

Module Federation can be crucial in the development of Micro-Frontends architectures since it resolves out-of-the-box a significant portion of the complexity generated by other alternatives (Taibi & Mezzalira, 2022). Module Federation can be used in Micro-Frontends both with client side composition and server-side composition, as it supports SSR (Zack Jackson, 2023b).

4.5.2 External Features

Jackson (2022) recommends using Module Federation to integrate functionalities developed by different teams that appear across multiple user flows or applications. However, this suggestion is primarily valid if these modules adhere to the Component Level Ownership pattern (CLO).

The Component Level Ownership pattern delegates as much responsibility as possible to the component itself, and it is based on four principles:

- *Smart Components* - These components should function almost independently, bearing all the necessary business logic and data requests;

- *Colocation* - The code should be well-organized, easily comprehensible, maintainable, and resilient. Changing a component's logic shouldn't inadvertently affect other business logic streams;
- *Loosely Coupled* - Components should minimize dependencies on each other, especially high-level modules. While some coupling is inevitable because components communicate with one another, minimizing these dependencies is the essence of being loosely coupled (Fowler, 2001; Jackson, 2022). This concept relates to the concept of modularity, described in chapter 3;
- *Ownership boundaries* - Different teams should clearly own and be responsible for different components. This ownership eases applications maintenance.

This pattern must take granularity into account, without the need to become super granular and delegate maximum responsibilities to all components. Jackson (2022) uses an example of a "title" component, which does not need to use the CLO pattern. However, for certain more complex features, this paradigm makes more sense.

4.5.3 Global Components

One potential use of Module Federation lies in managing global components. These components, found in numerous applications, are typically self-contained and should ideally remain consistent across various applications, making them strong candidates for the implementation of Module Federation. A more common alternative to sharing modules through Module Federation would be to update the component, publish it on a *Package Registry*, and then update all versions of the applications using that component. However, this method is more time-consuming

and labor-intensive for development teams (Jackson, 2022).

4.5.4 Polyliths

Polylith is a software architecture conceived by Joakim Tengstrand. This architecture decouples backend code into small, reusable parts that can be shared across various services. These "LEGO-like" components are designed to simplify the development of tools and services. By providing a certain level of abstraction, they allow developers to easily understand, compose, reuse, and exchange these blocks. A primary advantage of this architecture is the significant reduction of duplicate code through component sharing. (Polylith, 2022).

This architecture defines seven building blocks:

- *Function* - Functions form the foundational layer of this architecture. Within a Polylith system, most interactions occur through functions.
- *Library* - Libraries are segments of code archived in a versioned file. They can be located and downloaded from repositories, such as Maven Repository.
- *Component* - Components denote parts of the business domain, elements of infrastructure (like authentication and databases), or integrations with external systems.
- *Base* - Bases are a type of building block that expose their functionalities through a public API, be it GraphQL, Google Remote Procedure Call (gRPC), Representational State Transfer (REST), among others.
- *Brick* - This term is used interchangeably for both a Base and a Component.

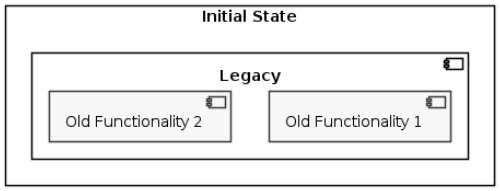
- *Project* - This identifies the Bricks and Libraries to be included in an artifact, such as a service, command-line tool, or a new library. It accentuates the principle of reusing components across diverse projects.
- *Development Project* - A project where libraries, components, and bases are all interconnected.
- *Workspace* - It signifies the place where all the building blocks are stored and where configurations for various projects are made.

Polylith is inherently language-agnostic, making it compatible with languages like JavaScript. In this context, Module Federation can be important, since it can assist in resolving and minimizing imported dependencies and streamlines the sharing of modules. (Jackson, 2022; Zack Jackson, 2023b; Polylith, 2022).

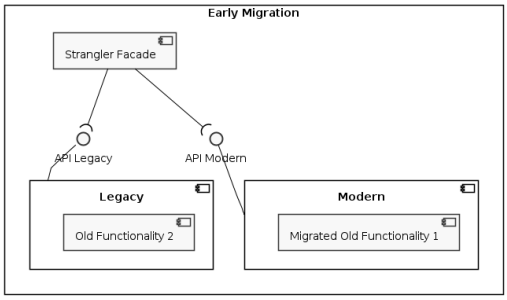
4.5.5 Strangler Pattern

Fowler (2004) coined the term "Strangler Application" or "Strangler Fig Application" as a metaphor used to describe a strategy for rewriting significant systems. This design pattern allows the existing application to be incrementally "strangled" as migration occurs, ensuring minimal impact on the current system. This approach has become especially popular for transitioning from monolithic systems to microservices architectures. The core idea is to develop a new system that seamlessly integrates with the old one. As new functionalities are added, the new system gradually "strangles" or replaces the old system. One of the primary benefits of employing this pattern is the ability to introduce new features while simultaneously migrating older functionalities in a phased manner (Fowler, 2004; Microsoft, 2023; Richardson, 2023). Figure 4.6 illustrates an example of migrating from an old

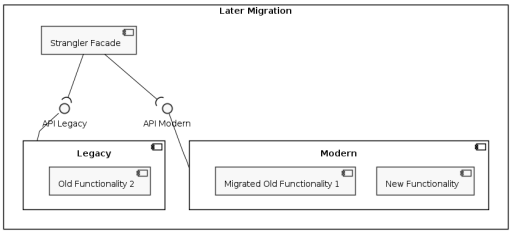
system to a new one using this strategy.



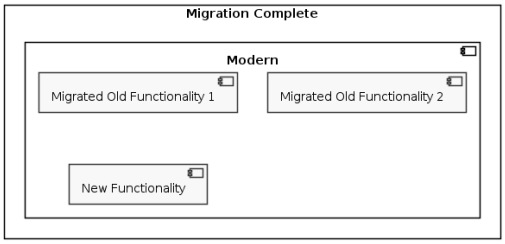
(a) Initial State



(b) Beginning of Migration



(c) Final of Migration



(d) Migration Complete

Figure 4.6: Example of a migration of a legacy system to a new system using the Strangler pattern. Adaptado de (Microsoft, 2023; Richardson, 2023)

There are two ways through which Module Federation can assist in the process of strangling a system:

- Converting New Microservices or Modules to Remotes of the Legacy System - This allows for the gradual strangulation of the legacy system by turning new micro-services or modules into “remotes” of the older system.
- Inverting the Strangling Process of the Legacy Service - This is achieved by creating a new service that imports modules from the legacy service, allowing it to function as a “remote” for the new service (host). When the Strangler

Pattern is employed for migrating to microservices, this method can ease the integration of shared code that is still present in the legacy system but is needed by the new microservices.

Both options allow for the reuse of modules from either the new or the legacy system, facilitating the service migration (Zack Jackson, 2023b).

4.5.6 A/B Testing

A/B testing, commonly known as split testing, involves comparing two versions of a webpage or app against each other to determine which one performs better in terms of achieving a desired action (like clicks, sign-ups, or purchases). As a business grows, its A/B tests might become more intricate, involving multivariate testing (testing more than two versions) (Unbounce, n.d.; Walsh, 2019).

Module Federation allows to serve multiple versions of modules to different user segments. For example, it is possible to serve two different versions of a module (Version A and Version B) to different user groups dynamically. This means that when User X visits the website, they could be served Version A of a module, whereas User Y might see Version B. With Module Federation, developers can push real-time updates to a A/B tested modules without disturbing the rest of the application, streamlining the process and reducing potential errors or downtime (Steyer, 2023; Unbounce, n.d.).

4.5.7 Server-Side Rendering

Server-Side Rendering (SSR) is a rendering technique that allows delivering fully rendered pages from the server to the browser. Unlike Client Side Rendering

(CSR), where the browser is responsible for rendering the entire webpage, SSR sends ready-to-render HTML response, allowing the browser to display the content immediately. This process occurs in mere milliseconds, and the immediate rendering of pages in SSR not only reduces the waiting time for users but also eliminates the initial blank page often encountered in CSR. SSR also offers an advantage for Search Engine Optimization (SEO), as search engines can more efficiently crawl and index the content of this sites (Jartarghar, Salanke, R, S, & Dalali, 2022). The use of Module Federation in conjunction with SSR allows to have the same advantages that module Federation offers to CSR applications but in SSR powered web applications (Zach Jackson, 2021, 2021). This integration also allows to have multi-server rendering, where remotes accept properties from the host application and perform the render at their origin, sending back the pre-rendered HTML. The use of Module Federation allows to offload the rendering load to another computer and this concept of distributed parallel rendering implies that Module Federation can be used in Multi-Threaded computing since the processing of rendered components is being done by using different CPUs. This distributed SSR can enhance scalability, enhance speed and reduce runtime overhead (Jackson, 2020; Zach Jackson, 2021).

4.6 Testing

In this section will be discussed some solutions to test the modules that use Module Federation in order to make sure the interfaces are not broken and the ability to import modules or files between independently compiled and deployed bundles at runtime is not affected.

4.6.1 Federated Unit Tests

In order to test applications using Module Federation, Zack Jackson (2021) defines a new approach to unit test in the form of Federated Unit Tests. This approach involves federating modules into unit tests in a manner analogous to federating them into applications. In a scenario where a file containing a Form imports a Button using Module Federation, the objective would be to test both the components, each of which is supplied by its own webpack build. This implies a create a new webpack configuration file in order to create the build to be used in the tests. The Listing 4.4 represents the Module Federation configuration for the Form component (Zack Jackson, 2021). The Module Federation configuration for the other components to be test are similar to the one presented in Listing 4.4.

```
1     new ModuleFederationPlugin({
2       name: 'form_app',
3       filename: "remoteEntry.js",
4       library: {type: "commonjs-module", name: "form_app"},
5       remotes: {
6         "dsl": reunited(path.resolve(__dirname, '../dsl/dist-test/
              remoteEntry.js'), "dsl")
7       },
8       exposes: {
9         "./Form": "./federated-cross-test/form.js"
10      },
11      shared: {
12        react: deps.devDependencies.react,
13        "react-dom": deps.devDependencies["react-dom"]
14      }
15    })
```

Listing 4.4: Example of a configuration of a webpack.test.config.js file for a federated unit test, retrieved from Jackson (2021).

The approach exemplified in Figure 4.4 relies on creating a test build for each repository/project, acting as a commonjs server build that remotes would create to expose their features as commonjs. The consumer of both this exposed applications will be a new project that acts as a host application but instead makes use of webpack's asynchronous capabilities to import federated modules and test them. The testing process is executed by running Jest against a webpack built test of test files. The webpack configuration for the test application acting as a host for the exposed remotes is described in Figure 4.5 (complete file configuration can be found in the attachments).

```
1           new ModuleFederationPlugin({
2             name: "test_bundle",
3             library: {type: "commonjs-module", name: "test_bundle"},
4             filename: "remoteEntry.js",
5             exposes: {
6               "./render":"./test/suspenseRender.js"
7             },
8             remotes: {
9               "form_app": reunited(path.resolve(__dirname,'../form_app/
                    dist/test/remoteEntry.js'),"form_app"),
10              "dsl": reunited(path.resolve(__dirname,'../dsl/dist/
                    remoteEntry.js'),'dsl')
11            }
12          }),
```

Listing 4.5: Example of a partial configuration of a webpack.test.config.js file for a federated unit test test application, retrieved from Jackson (2021).

This process would allow for tests to be written as presented in Figure 4.6.

```
1   const Form = import("form_app/Form");
2   const Button = import("dsl/Button");
3
4   describe("Federation", function () {
5     it("is rendering Nested Suspense",async()=>{
6       const from = await Form
7       console.log(await suspenseRender(from.default))
8     })
9     it("Testing Button from Remote", async function () {
10      const Btn = (await Button).default
11      const wrapper = render(<Btn/>);
12      expect(wrapper).toMatchSnapshot()
13    });
14    it("Testing Button from Form", async function () {
15      const Frm = (await Form).default
16      const wrapper = mount(<Frm/>);
17      expect(wrapper).toMatchSnapshot()
18    });
19  });
```

Listing 4.6: Example of federated unit test, retrieved from Jackson (2021).

For this particular case the different projects are locally being build and tested but, in the context of a Continuous Integration/Continuous Delivery (CI/CD) environment, the test container can pull down other repositories or buckets and execute them locally. In the case of a mono-repository, this tests could be done

either locally or in CI/CD environments without the need to pull down other repositories or buckets (Zack Jackson, 2021).

4.6.2 Smoke Tests

Smoke testing, also known as Build Verification Testing, is a high-level type of testing conducted to make sure that the basic functions of a program are working, ensuring that the application under test is operational and has its core functionalities working as expected, while not bothering with finer details (Cannavacciuolo & Mariani, 2022; Chauhan, 2014; Herbold & Haar, 2022). The origins of smoke testing can be traced back to hardware and plumbing industries, where it was used to detect any blatant issues or breaks in the system (Chauhan, 2014).

In the context of Module Federation smoke testing can be used to determine the stability of the software build as it acts as a preliminary check to ensure the every component of the system under test and interactions between them are operational. This type of tests can be useful for checking if different modules are correctly integrating and communicating with each other. If a module is importing components from another module, the smoke test checks if this operation is successful and doesn't break the system under test.

4.7 Alternatives

This chapter explores other techniques and approaches that can be used instead of Module Federation. The discussion describes the strengths and weaknesses of these alternatives, offering insight into the options available for managing and optimizing

web component integration and library sharing.

4.7.1 Import Maps

Import maps are a JSON object that allows, when importing Javascript modules, to control how the browser resolves module specifiers. It maps the text in a module specifier to a specific value, ensuring the JSON object follows the Import map JSON representation format. (Mozilla, 2023c). Import Maps have emerged as a robust tool, offering a new paradigm for orchestrating micro-frontends at runtime enabling the redefinition of module specifiers, providing a significant degree of flexibility and broad applicability across various technologies. Import Maps are not confined to any specific technology or build tool, making them a versatile choice for projects with diverse technological stacks. This flexibility extends to dynamic imports and version handling, allowing for structured management of different module versions through scopes (Steyer, 2022b; Zaikin, 2023) The Listing 4.7 provides a example on how to create an import map to share a library.

```
1    <script type="importmap">
2    {
3        "imports": {
4            "date-fns": "./libs/date-fns.js"
5        }
6    }
7    </script>
```

Listing 4.7: Example of usage of import maps to share a library, retrieved from Steyer, 2022b.

Import Maps efficiently manage external dependencies through the import maps configuration in the HTML. This approach contrasts with other methods that handle dependencies at build time, offering a more streamlined and efficient process for dependency management. The external dependencies can also be loaded in different versions as Import Maps offers the possibility to have multiple versions of the same dependency. For the resolution of this version conflicts import maps offer so-called scopes. The Listing 4.9 exemplifies the usage of scopes to deal with version mismatch (Steyer, 2022b; Zaikin, 2023).

```
<script type="importmap">
{
    "imports": {
        "date-fns": "./libs/date-fns.js",
        "is-bridging-day": "./js/is-bridging-day.mjs"
    },
    "scopes": {
        "/js/is-bridging-day.mjs": {
            "date-fns": "./libs/other-date-fns.js"
        }
    }
}
</script>
```

Listing 4.8: Example of usage of import maps to share a library, retrieved from Steyer, 2022b.

Imports Maps also offer the possibility to have dynamic import maps. They enable the management of external dependencies without necessitating a bundler. However, the bundler is needed to include the corresponding import statements in the bundle on a one-to-one basis, rather than including the referenced files within the bundle as well. This approach is commonly referred to as externals in most bundlers. Typically, such externals can be delineated through the configuration of the bundler. Listing 4.9 exemplifies the usage of externals making use of esbuild (Mozilla, 2023c; Steyer, 2022b; Zaikin, 2023).

```
1     await esbuild.build ({
2         entryPoints: ["js/is-bridging-day.mjs"],
3         external: [ "date-fns" ],
4         format: "esm" ,
5         target: [ "esnext" ],
6     });
```

Listing 4.9: Example of usage of version mismatch resolution with import maps, retrieved from Steyer, 2022b.

Steyer (2022a) coined the term Native Federation as a "Framework and tooling-agnostic implementation of Module Federation". This tool abstracts the complexity of import maps and creates a similar interface to the interface provided by the Module Federation Plugin. Although this solution remains in its BETA phase, it waits integration with Angular after experimental esbuild-based builder is used. Listing 4.10 demonstrates the utilization of the @angular-architects/native-federation library.

```
1   const { withNativeFederation, shareAll } = require('
        @angular-architects/native-federation/config');
2
3   module.exports = withNativeFederation({
4       name: 'mfe1',
5       exposes: {
6         './Module': './projects/mfe1/src/app/flights/flights.
              module.ts',
7       },
8       shared: {
9         ...shareAll({ [...] }),
10      },
11  });
```

Listing 4.10: Example of usage of Native Federation plugin, retrieved from Steyer, 2022b.

The Table 4.1 describes the principal diferences between the usage of Import Maps and Module Federation.

Table 4.1: Comparison between Module Federation and Import Maps.

	Module Federation	Import Maps
Browser Support	Dependent on Webpack support (Webpack supports all browsers ES5-compliant)	Cross-Browser Support only for newer

Table 4.1: Comparison between Module Federation and Import Maps.

	Module Federation	Import Maps
Configuration Complexity	Simple Configuration	Provides Little Abstraction (Native Federation Plugin can reduce this complexity)
Build Tool Dependency	Webpack	-
Angular Integration	Simple Integration using Module Federation Angular Architects Plugin	Needs manual configuration
Version Mismatch	Resolves last compatible version	Needs to be manually resolved

4.7.2 Single-Spa

Single-Spa is a framework that takes on the responsibility of routing, loading, and unloading applications based on the location provided. It allows the integration of multiple frameworks and versions making it a suitable choice for migrating from a legacy system to a modern one, allowing incremental upgrades. Implementing Single-Spa involves configuring the root project to manage child applications. Each application, whether a micro-frontend or a utility, is treated as an independent unit with its lifecycle events. Single-Spa provides the necessary hooks to manage these lifecycles (Single-Spa, 2023b, 2023a). The usage of Single-Spa is not a direct alternative to Module Federation, instead it can also be used in conjunction with

it or with Import Maps (Single-Spa, 2023c).

4.8 Case studies

Zack Jackson (2023a) lists part of the companies that are using module federation to build their applications, this companies list includes Cisco, Amazon, Shopify, etc. The grand majority of this companies implementation and case studies are not public and information could not be retrieved about it (Zack Jackson, 2023a). This chapter will describe some of the companies using module federation, taking into consideration the limited information available to the public.

4.8.1 Netflix

Netflix's felt the need for a flexible and scalable solution and the Revenue and Growth Tools (RGT) team introduced Lattice, a framework designed for micro frontends. This framework provides an abstraction layer for React web applications to leverage, and allows to resolve external dependencies on-demand from any number of sources (Possumato, Tomlin, Andree, Shim, & Pilani, 2021). Lattice was created to fit in five main objectives:

- **Low Friction Adoption** - Foster reuse of frontend code using standard React methods.
- **Weak Dependencies** - Allow hosts to access modules from internal remote bundles via HTTPS, compliant with standards like Webpack Module Federation or native JavaScript Modules.
- **Alignment & Flexibility** - Ensure plugins align with Netflix standards and

deliver core functions without excess baggage.

- **Metadata-Driven** - Use configuration-based plugins for dynamic application adaptation.
- **Rapid Development** - Streamline the development process by negating frequent builds and deployments. By using TypeScript declarations and well-defined interfaces, both plugins and host applications can be developed simultaneously.

The integration of the Lattice framework can be visualized in Figure 4.7.

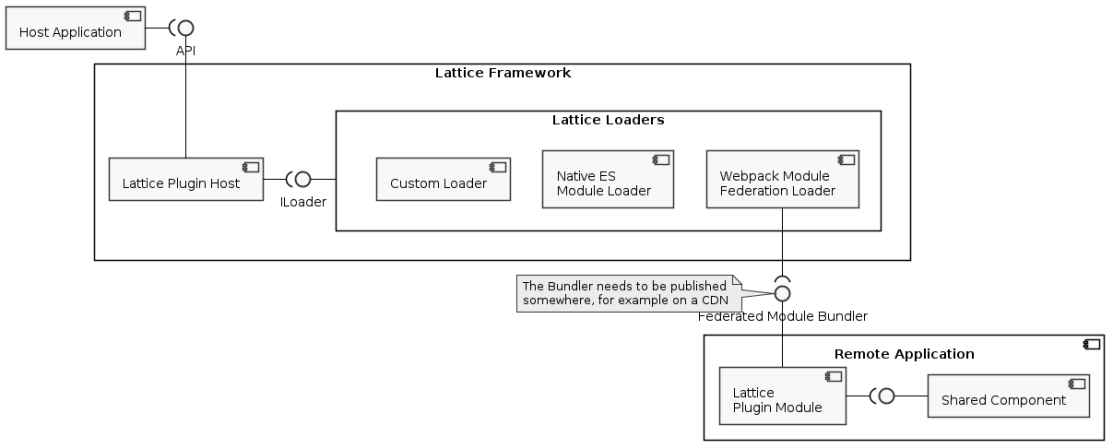


Figure 4.7: Lattice integration with Module Federation. Adapted from Possumato, Tomlin, Andree, Shim, & Pilani, 2021.

The host application relies on a set of plugins to asynchronously load external modules, Webpack Module Federation Loader being one of them. When using Lattice the remote modules needs to export method signatures that match Lattice Plugin Module interface.

4.8.2 Telia

Telia Company is a multinational telecommunications company that originates from Sweden and Finland. Telia main motivation behind module federation adoption was to merge two large monolithic React applications. The implementation consists of a main federated module and several other micro frontends housed in a mono repository. The main federated module is responsible for setting the global layout of the user interface, such as the global navigation menu and top bar. It also determines which micro-frontend should be rendered based on the route. Each micro frontend then handles its internal routing. Additionally, the main federated module serves as a shared library for all applications, offering shared layout components and formatting utilities. It also handles user authentication, sets up an Apollo client for data fetching, and provides global state to the entire system (Grini, 2021; Telia, 2023).

4.8.3 Housing

Housing is a platform based in India that serves as an advertisement hub for homeowners, landlords, developers, and real estate agents. The company successfully leverages module federation to manage its micro-frontend applications. Additionally, they were successful in implementing module federation for their server-side application and integrating it into their architecture (Housing, 2023; Saini, 2023).

4.8.4 Rivian

Rivian, an American automobile manufacturer, specializes in creating electric vehicles. The modules managed by webpack aren't restricted to just JavaScript,

they can be of any type (Java, C, etc.), as long as Node or Webpack can comprehend them, they can be transmitted using the federation delivery protocols. This allows Rivian to have the vehicle's onboard and infotainment systems, as well as all user interfaces, utilize and being managed by Module Federation. This technology facilitates communication with the micro-controllers and drives the interfaces that the users interact with (devtools-fm, 2022; Rivian, 2023).

4.9 Summary

This chapter provided an overview of Module Federation functionalities, its diverse applicability, and testing methodologies. It was explored alternatives to this technology and presented real examples from companies implementing solutions using Module Federation.

Chapter 5

Implementation

This chapter describes the practical aspects of the project. It is explained the chosen technologies and the specifics of Module Federation implementation. Lastly it is discussed the integration challenges and solutions in a local development environment.

5.1 Technological Decisions

The technological choice around the use of Module Federation was influenced by its ease of use and compatibility with the project's existing infrastructure. At the beginning of the implementation, alternatives like import maps were not supported by iOS mobile browsers, presenting a significant limitation since part of our users use this type of device and browser. Additionally, the project was already using Webpack for the host and all web components, facilitating the integration with Module Federation. The option of using esbuild was discarded as its compatibility

with Angular, which the web components are built on, is still in the beta phase, posing potential risks and uncertainties.

The type of tests selected to use in this project were smoke tests using cypress which decisions are out of the scope of this thesis. This decision was made due to the ease of implementation of smoke tests and knowledge of the team regarding this type of testing tool, ensuring efficient and effective testing without extensive setup or complexity. This approach aligns with the project's scope and objectives without unnecessary overhead.

5.2 Module Federation

The migration from basic web components to federated web components occurred in two distinct steps. The initial step involved the migration of the web components (see sub-chapter 2.1.4). Subsequently, the second step involved migrating the widgets to make use of Module Federation (refer to sub-chapter 2.1.5). The Angular-Web-Componentes are always present in the MWP Client application and therefore they are always loaded at the start. In contrast, the widgets' loading is dependent on the configuration of the dashboards in which they present, requiring dynamic imports.

5.2.1 Angular-Web-Components

The examples presented in this subsection originate from a single Angular-Web-Component. However, this approach was replicated across all Angular-Web-Components. For these components, the `bootstrap.ts` file is exposed, which

in turn imports the main.ts file. This ensures that the remotes are loaded initially, followed by the loading of the application itself, as illustrated in Listing 5.1. For the `bootstrap.ts` file, the `@angular-architects/module-federation-tools` was used to abstract and reduce the complexity involved in the bootstrap of the Angular-Web-Component's application.

```
1   // main.ts
2   import('./bootstrap').catch(err => console.error(err));
3
4   // bootstrap.ts
5   import { bootstrap } from '@angular-architects/module-
        federation-tools';
6   ...
7   bootstrap(AppModule, {
8       production: environment.production,
9   });
10  ...
```

Listing 5.1: Implementation of the entry file for the Angular-Web-Component.

The Angular-Web-Components use a `webpack.config.js` configuration, as represented in Listing 5.2. This configuration file sets the library type to 'global', allowing webpack to interpret the federated module as a global variable within the `globalObject`. This global object is named "idCard", and its entry file is designated as `id-card.entry.js`. The Angular-Web-Component exposes the `bootstrap.ts` file, which includes all the component's code. Given that all the Angular-Web-Components reside within an Nx Monorepo, they inherently share identical dependency versions, and for managing shared dependencies it was used

the `@angular-architects/module-federation` library. This library allows setting the `requiredVersion` to `auto`, enabling the extraction of the dependency value directly from the global `package.json`.

```
1   const { share } = require('@angular-architects/module-
        federation/webpack');
2   ...
3   new ModuleFederationPlugin({
4     name: 'idCard',
5     filename: 'id-card.entry.js',
6     library: {
7       type: 'global',
8       name: 'idCard',
9     },
10    exposes: {
11      '.': 'apps/web-components/id-card/src/bootstrap.ts',
12    },
13    shared: share({
14      '@angular/core': {
15          singleton: true,
16          strictVersion: true,
17          requiredVersion: 'auto'
18      },
19      // Omitted for brevity.
20    }),
21  }),
```

Listing 5.2: Module Federation Plugin configuration for Web Component ID Card.

In order for the exposed web components to be consumed by the Host application MWP Client, the `webpack.config.js` file needed to be updated as illustrated in Listing 5.3. Serving as the entry point and host application, the MWP Client ensures that all dependencies used across the web components are loaded eagerly. This means that the Module Federation plugin delivers the modules synchronously rather than placing them in an asynchronous chunk. Such a setup permits the utilization of these shared modules in the initial chunk. Within this configuration file, an `remotes` object is also defined, listing all the remote names and the corresponding URLs where the entry file is situated.

```
1     new ModuleFederationPlugin({
2         name: 'mwpClient',
3         remotes: {
4             idCard: `idCard@https://some-place.net/id-card.entry.js`,
5         },
6         shared: {
7             "@angular/core": {
8                 singleton: true,
9                 strictVersion: true,
10                requiredVersion: deps["@angular/core"],
11                eager: true,
12            },
13            // Omitted for brevity.
14        },
15    })
```

Listing 5.3: Module Federation Plugin configuration for MWP Client.

5.2.2 Widgets

The widgets are designed to load within the “Dashboards” Angular-Web-Component. Given that the widgets are specific for each individual dashboards, it was essential

to load these widgets dynamically, rather than preloading them as it was done with other web components. To achieve this, a new Angular component was introduced within the dashboards, as depicted in Listing 5.4.

```
1  export class WidgetComponent implements AfterViewInit {
2      @ViewChild('tileWrapper', { read: ElementRef, static: true })
           tileWrapper!: ElementRef;
3      @Input() widget!: Widget;
4
5      ngAfterViewInit(): void {
6          this.loadWidget();
7      }
8
9      private loadWidget(): void {
10         this.retryOperation(() =>
11             this.widgetService.loadRemoteScript(this.widget.
                   remoteEntryConfig), 1000, 1)
12         .then(() =>customElements
13             .whenDefined(this.widget.remoteEntryConfig.elementTagName
                   )
14             .then(async () => {
15                 const widgetElement = await this.createWidgetElement
                       ();
16                 this.widgetElement = widgetElement;
17                 this.applyWidgetContext();
18                 this.tileWrapper.nativeElement
19                 .appendChild(widgetElement);
20                 if (!this.widgetElement.shadowRoot) {
21                     this.createShadowDomContainer();
22                 }
23         })).catch(() => { // Omitted for brevity. });
24     }
25 }
```

Listing 5.4: Widget Component to load widgets dynamically.

Widget loading via Module Federation is facilitated using the @angular-architects/module-federation library. However, certain widgets may not be ready for importation through module federation. To address this, a "moduleFederationReady" property was incorporated into the widget configuration within the database. If this property is set to `true`, the widget will be loaded using the Module Federation approach. Conversely, if a widget isn't configured for module federation, it reverts to the previously established import method. This strategy ensures backward compatibility with older widgets that are not ready for Module Federation. The procedure for loading widget bundles is detailed in Listing 5.5.

```
1     loadRemoteScript(remoteEntryConfig: WidgetRemoteEntry): Promise<
          unknown> {
2         if (!remoteEntryConfig.moduleFederationReady) {
3             if (!(scriptAlreadyLoaded() ||
                  customElementAlreadyDefined())) {
4                 return this.loadWidgetNotModuleFederationRemote(
                      remoteEntryConfig.remoteEntry);
5             }
6             return Promise.resolve();
7         }
8         return loadRemoteModule({
9             remoteEntry: remoteEntryConfig.remoteEntry,
10            remoteName: remoteEntryConfig.name,
11            exposedModule: './web-components',
12        });
13    }
14 }
```

Listing 5.5: Load of remote scripts.

Widgets not utilizing Module Federation are loaded as outlined in Listing 5.6.

Each of these widgets consists of a singular JavaScript file, which is subsequently appended to the document's head.

```
1     private loadWidgetNotModuleFederationRemote(remoteEntry: string):
          Promise<void> {
2         return new Promise((resolve, reject) => {
3             if (!moduleMap[remoteEntry]) {
4                 const script: HTMLScriptElement = document.
                      createElement('script');
5                 script.type = 'text/javascript';
6                 script.src = remoteEntry;
7                 script.async = true;
8                 script.onload = (): void => {
9                     moduleMap[remoteEntry] = true;
10                    resolve();
11                };
12                script.onerror = reject;
13                document.getElementsByTagName('head')[0].appendChild(
                      script);
14            } else {
15                resolve();
16            }
17        });
18    }
19 }
```

Listing 5.6: Dynamic load of widgets not using Module Federation.

To enable the Dashboards Angular-Web-Component to import widgets utilizing Module Federation, these widgets must make their code accessible. Listing 5.7 illustrates the `webpack.config.js` file for the Timeline Widget in the context of Module Federation. This configuration is similar across all widgets.

```
1       new ModuleFederationPlugin({
2           library: {
3               type: 'var',
4               name: 'widgetsTimeline',
5           },
6           name: 'widgetsTimeline',
7           filename: 'timeline.remoteEntry.js',
8           exposes: {
9               './web-components': './apps/widgets/timeline/src/
                    bootstrap.ts',
10          },
11          shared: SharedDeps,
12      }),
13  }
```

Listing 5.7: Module Federation Plugin configuration for Timeline Widget.

5.2.3 iFrames

The integration of iFrames with Module Federation is not an usual use case and presents several challenges. An initial solution to the issue of library sharing within embedded iFrames was the utilization of the `postMessage()` method, available on both the parent window and the iFrame window object. This method stands as the most secure mechanism for facilitating two-way communication between distinct iFrames. However, when considering the global objects that webpack employs for communication and cache management. Objects such as ___webpack_modules___, the global array (webpackChunk), and ___webpack_share_scopes___ are complex and, in many cases, not JSON serializable. This non-serializability implies that these objects cannot be shared between iFrames using the Window Messaging API.

Given these constraints, one viable approach to object sharing between two iFrames, without using the Messaging API, is to host both the application and the iFrame under the same domain. While this might not be universally applicable, it is a feasible strategy for Core Apps where teams have full ownership and can ensure both apps are in the same domain. With this perspective, a demonstrative example was developed using the MyWorkplace Shell as host and the App Management within an iFrame, the sequence diagram in Figure 5.3 is used to illustrate the implemented process.

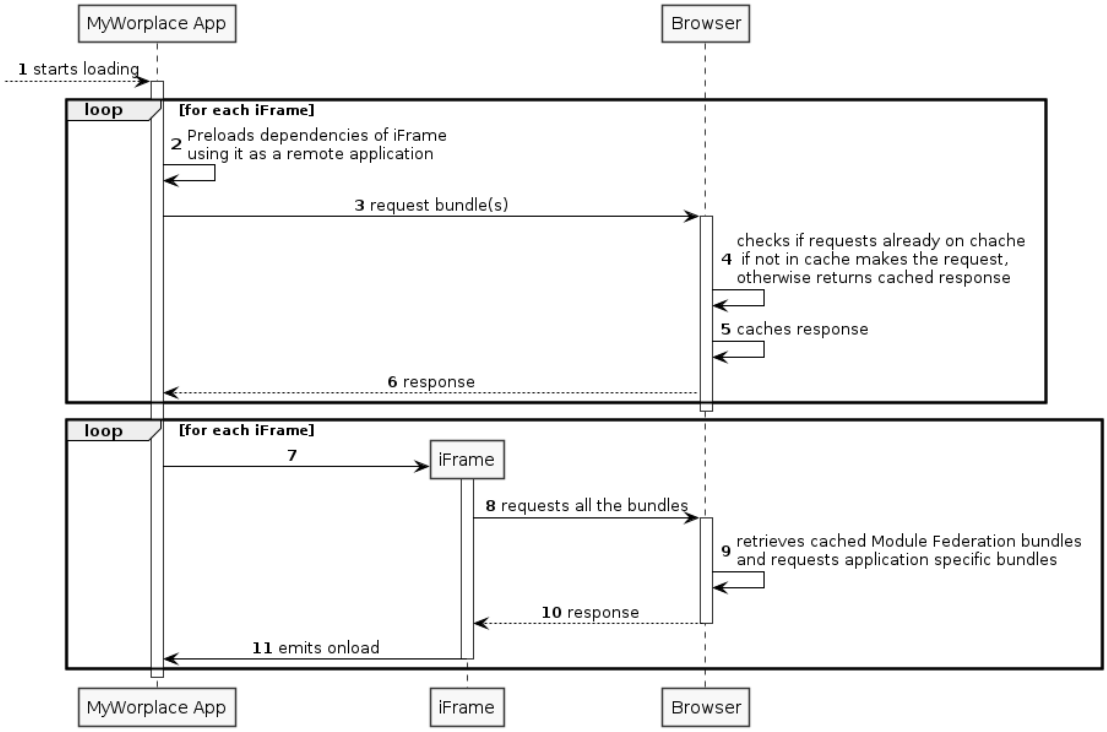


Figure 5.1: iFrame Loading flow sharing dependencies.

The procedure must be initiated prior to the loading of the Host application. If the iFrames reside in distinct sub-domains within the primary domain, they should

be loaded in advance of the main host application (if all the applications are under the same full domain the browser will handle the cache of the bundles). Listing 5.8 provides an illustration of the Module Federation configuration within the Host application's `main.ts` file.

```
1   preLoadIFrames()
2       .then(() => import('./bootstrap')
3       .catch(err => console.error(err)));
4
5   async function preLoadIFrames(): Promise<void> {
6       const iframes = await getIFrames();
7
8       for (const iframe of iframes) {
9           await loadIFrameDependencies(iframe.url, iframe.
                scope);
10      }
11  }
```

Listing 5.8: Main file configuration within the Host application loading iFrames

The getIFrames method can retrieve a list of iFrames for preloading from various sources. This list can be sourced from an API endpoint, global window variables, or even be a static list designated for preloading. The `loadIFrameDependencies` method is subsequently invoked, with its implementation depicted in Figure 5.9.

```
async function loadIFrameDependencies(url: string, scope:
    string): Promise<void> {
    await loadScript(url);
    await __webpack_init_sharing__('default');
    const container = window[scope];
    await container.init(\__webpack_share_scopes__.default)
}
```

Listing 5.9: Dynamic loading of container for iFrame applications

The `loadScript` method injects the entrypoint script into the document, allowing the loading of iFrame dependencies. The host application references these preloaded dependency bundles, preventing redundant loading. However, this approach will only work for specific use cases, therefore was not used in real applications and was implemented primarily for experimental purposes. Alternative methods, where the iFrame attempts to access the parent frame's global objects to verify if the bundles were previously loaded, were explored and unfortunately, these alternatives were not successful. Had they been successful, there would have been a potential to bypass even the browser cache, because all the process would then be handled by Module Federation itself.

5.2.4 Non Javascript Files

The type of files that can be shared through Module Federation is limited only by Webpack's ability to interpret them. These files can be shared between remotes and host applications. While teams can create custom loaders based on their specific requirements, there is already several pre-existing file loaders. These include loaders

for compiling Rust into Web-Assembly, processing images, fonts, styles, and more. In this context, this section will focus on the sharing of CSS styles. The Module Federation configuration for the Angular-Web-Component Dashboards, which acts as a host, is illustrated in Listing 5.10.

```
1   module: {
2     rules: [
3       {
4         test: /federated\/.*\.css$/,
5         use: ['css-loader'],
6       }
7     ]
8   },
9   plugins: [
10    new ModuleFederationPlugin({
11      library: {
12        type: 'global',
13        name: 'dashboards',
14      },
15
16      name: 'dashboards',
17      filename: 'dashboards.entry.js',
18      exposes: {
19        '.': 'apps/web-components/dashboards/src/bootstrap.ts',
20        './styles': 'apps/web-components/dashboards/src/app/federated
              /styles.css'
21      },
22
23      shared: SharedDeps,
24    }),
25  ],
```

Listing 5.10: Module Federation configuration for the Dashboards Angular-Web-Component for css file sharing

The styles intended for sharing are located within the "federated" folder (just for organization, could be placed anywhere). These styles utilize the css-loader plugin to transform CSS files into JavaScript, which can then be imported by the remote application. The styles.css file is subsequently shared for consumption by the remote application. Listing 5.11 showcases the Module Federation configuration file for the Translator Widget.

```
1     new ModuleFederationPlugin({
2       name: 'widgetsTranslator',
3       filename: 'translator.remoteEntry.js',
4       exposes: {
5         './web-components': './apps/widgets/translator/src/
              bootstrap.ts',
6       },
7
8       remotes: {
9         dashboards: `dashboards@http://localhost:4004/
              dashboards.entry.js`
10      },
11
12      shared: SharedDeps,
13    }),
```

Listing 5.11: Module Federation configuration file for the Translator Widget consuming dashboards Angular-Web-Component

In this specific scenario, the Translator Widget functions both as a remote, since it is loaded by the Angular-Web-Component Dashboards, and as a host, as it

consumes the styles provided by the Dashboards. In order to use the import `dashboards/styles`, it is necessary to first declare this module, as illustrated in Listing 5.12.

```
1  // app.component.d.ts
2  declare module "dashboards/styles";
```

Listing 5.12: Example of declaring module in Typescript

The styles must be injected into the component. Specifically, in the `app.component.ts` file, it is essential to inject the styles into the shadow dom, given that the Translator Widget is a web component. Listing 5.13 demonstrates the method by which the Translator Widget integrates the styles into the component.

```
1  import styles from 'dashboards/styles';
2  // omitted for brevity
3  export class AppComponent implements AfterViewInit {
4      private readonly el: ElementRef = inject(ElementRef);
5      ngAfterViewInit(): void {
6          this.injectStyles(styles.toString());
7      }
8      injectStyles(css: string): void {
9          const style = document.createElement('style');
10         style.appendChild(document.createTextNode(css));
11         this.el.nativeElement.shadowRoot.appendChild(style)
12     }
```

Listing 5.13: Integrating shared styles in the Translator Widget

Following the injection of the styles, the application can then make use of the styles provided by the dashboards Angular-Web-Component in the HTML templates.

This serves as a practical illustration of how to share non-JavaScript files using Module Federation. Within the MyWorkplace application, styles shared among various Angular-Web-Components and Widgets are stored in a CDN. This storage approach facilitates browser caching of the response, and therefore the current implementation is unnecessary for the time being.

5.3 Local Development Environment

In this sub-chapter, two distinct approaches for local development setup are compared. The traditional approach explains the setup before the introduction of Module Federation and the approach to local development environment using Module Federation.

5.3.1 Traditional Approach

The previous development process was somewhat cumbersome. Whenever the team needed to integrate Angular-Web-Components or Widgets from another repository into the client application, a multi-step process was required.

- Build of the web components had to be generated.
- These components were then manually copied into the host's repository.
- Client application was run, integrating the new components.

After any changes to the codebase of the web components in test, the component needed to be re-builded and copied again to the host repository. The Figure 5.3

illustrates the process described.

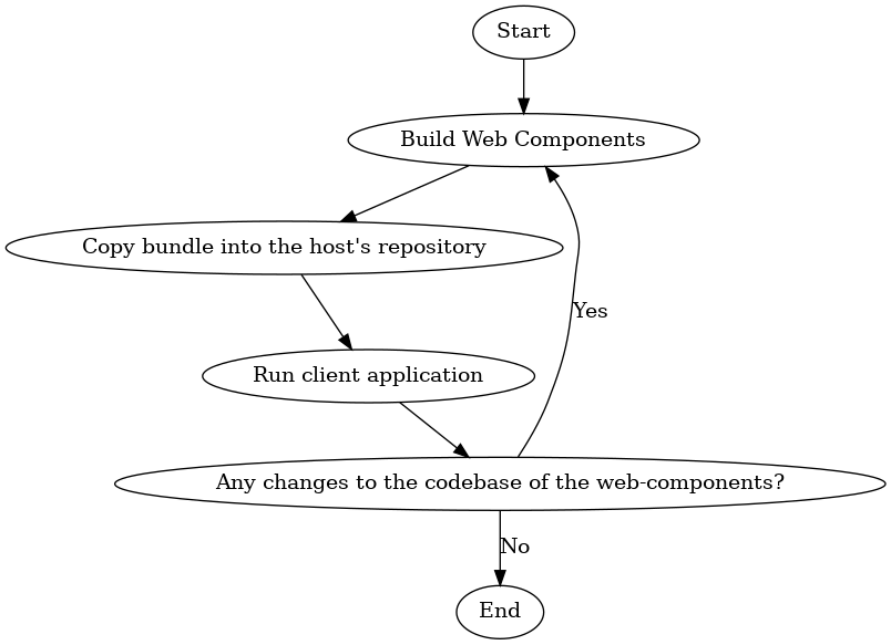


Figure 5.2: Traditional approach local development environment flow

This approach, while functional, was time-consuming and the repetitive nature of the process left ample room for improvement.

5.3.2 Module Federation Approach

Module Federation allows the tedious process of building and copying components to be eliminated. Instead, developers can directly point to a specific port where the components or modules are hosted. Host application can dynamically load these components at runtime, without the need for manual transfers or rebuilds. Now the process follows the following steps:

- Serve locally web component.
- Serve locally host application.

After any changes to the codebase of the web components, in order for the changes to be reflected in the host application its only needed to refresh the page of the host application. The Figure 5.3 illustrates the flow of local development using Module Federation.

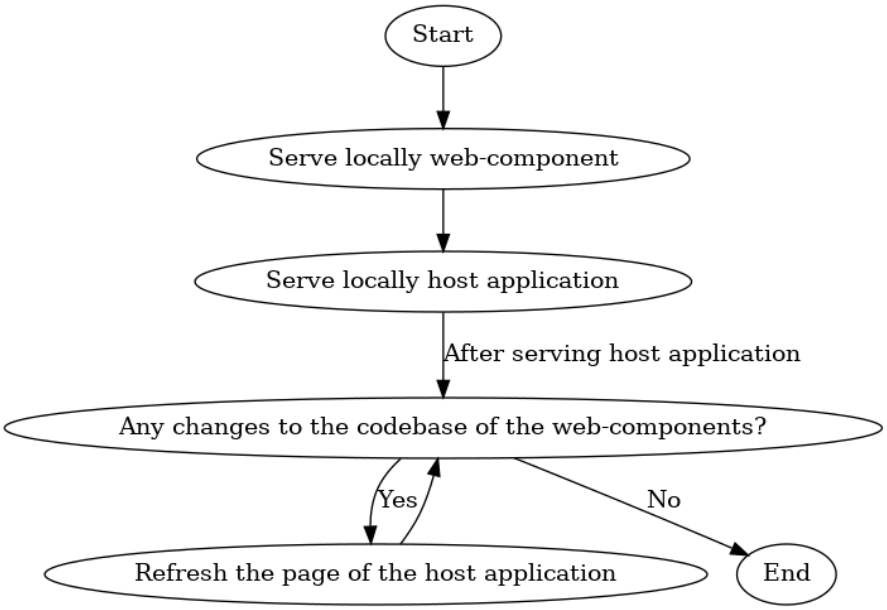


Figure 5.3: Module Federation local development environment flow

In order for this process to be abstract as possible to the developers, the configuration illustrated in Figure 5.14 was created in the host application. This allows for a simple and fast configuration of the web- components and widgets that should be run locally using Module Federation. Widgets and Angular-Web-Components not included in this configuration will be retrieved from the test environment.

```
1     const config: Configuration = {
2         ...configuration({
3             webComponents: {
4                 toRunLocal: [
5                     WebComponentsNames.APPS_MENU,
6                 ]
7             },
8             widgets: {
9                 toRunLocal: [
10                    WidgetsNames.WORLD_CLOCK,
11                    WidgetsNames.TIMER,
12                ]
13            },
14        })
15    };
16
17    export default config;
```

Listing 5.14: Configuration abstraction for setting up local environment with Module Federation

The abstract configuration will then be passed to the Module Federation plugin that will dynamically chose to run a web component from the test environment or from a pre designated localhost port. Figure 5.15 illustrates the code that allows this process to be dynamic and simple.

```
1   // webpack.config.ts
2   new ModuleFederationPlugin({
3       name: 'mwpClient',
4       remotes: remotes(webComponents),
5       shared: sharedDependencies,
6   }),
7
8   // remotes.ts
9   export const localDefaults: Map<WebComponentsNames, string> = new
        Map([
10      [WebComponentsNames.APPS_MENU, 'http://localhost:4003'],
11      ...
12  ]);
13
14  export const remotesDefault = new Map([
15      [WebComponentsNames.APPS_MENU, '/web-components/apps-menu'],
16      ...
17  ]);
18
19  export function remotes(webComponents: Map<WebComponentsNames,
        string> = remotesDefault) {
20      return {
21          appsMenu: `appsMenu@${webComponents.get(
                WebComponentsNames.APPS_MENU)}/apps-menu.entry.js`,
22          ...
23      };
24  }
```

Listing 5.15: Dynamic web component loading for local environment with Module Federation

This process makes use of the library `http-proxy-middleware` to create proxies. The Figure 5.16 illustrates how this library is being used to locally create proxies

to the locally run federated web-components.

```
1   function localWidgetsProxies(widgetsNames: string[]):
        RequestHandler[] {
2       const array: RequestHandler[] = [];
3
4       widgetsNames.forEach(name => {
5           const widgetLocalObject = Widgets[name];
6           array.push(
7           createProxyMiddleware(`/widgets/${widgetLocalObject
                .name}`, {
8               target: `http://localhost:${widgetLocalObject.
                    port}`,
9               pathRewrite: {
10              [`^/widgets/${widgetLocalObject.name}`]: `/`
11              },
12              secure: false
13          })
14          );
15      });
16
17      return array;
18  }
```

Listing 5.16: Proxies configuration to local running widgets using Module Federation

This configuration and process allowed the developers to be more productive and to waste less time in repetitive tasks.

5.4 Summary

In this chapter, the various significant aspects and stages of the project's execution were described. The technological decisions made were explained, providing insight into the choice of using Module Federation and the adoption of specific testing strategies. It is also detailed the implementation of Module Federation within the project, focusing on its application in Angular-Web-Components and widgets. It is also discussed the local development environment, contrasting the traditional approach with the Module Federation approach, allowing for a deeper understanding of the benefits and advancements brought after the integration of Module Federation.

Chapter 6

Experiments and Evaluation

This chapter describes systematic experiments conducted to assess the impact of various libraries being shared with Module Federation on application performance, particularly when shared across multiple applications. Through a combination of quantitative measurements, this chapter attempts to understand the interplay between library size, sharing mechanisms, and resultant performance metrics. The experiments have been created with the Angular-Web-Components and Widgets and analyzes the performance metrics in a controlled Test environment, highlighting the incremental sharing of libraries, by the means of Module Federation.

During this chapter, both Angular-Web-Components and Widgets will have their names referenced as acronyms due to the large length of their names. The full list of the mapping between these web components and their acronyms can be found in Table 7.1 in the Attachments.

6.1 Objectives

The primary aim of this chapter is to systematically evaluate the implications of integrating specific libraries into the Module Federation configuration. By conducting this analysis, the following objectives are set forth:

- Assessment of Impact: Understand the performance impact of sharing libraries in a range of metrics, such as bundle size, load time, and the number of network requests.
- Optimization Analysis: Identify potential areas of improvement and conclude if there are certain libraries that, when shared, offer positive or negative impact on the overall efficiency of the application.
- Strategic Library Inclusion: Provide a foundation for informed decision-making regarding which libraries should be inlcuded in the module federation configuration, based on their performance impact and relevance to the application's functionality.

These objectives will allow to take a choice on what libraries to share and to understand what are the consequences of using Module Federation in the current setup.

6.2 Methodology

In this analysis, the methodologies employed attempt to evaluate the integration of specific libraries into the Module Federation configuration as well as the impact on the number of applications sharing this libraries. This systematic approach has the following steps:

- Baseline Measurement: Before any modifications, a baseline measurement of the application's performance metrics was established. This served as a reference point against which subsequent changes could be compared against.
- Selection of Libraries: The libraries under consideration, namely @angular/-core, @angular/common and @enterprise-ds/components, among others, were chosen based on their size relative to the single bundles before using Module Federation;
- Iterative Integration: Libraries were added to the Module Federation configuration one by one, in a sequential manner. After the integration of each library, the application was tested to measure its impact on performance metrics. The same approach was taken to measure the differences after adding more and more web components using Module Federation;
- Performance Metrics Evaluation: Key performance indicators, such as bundle size, load time, and the number of network requests, were recorded after the addition of each library and after every addition of a web component. This allowed for a comparative analysis against the baseline measurements;
- Analysis of Results: The data obtained from the tests were analyzed to discern patterns, anomalies, and significant findings.

6.3 Library Size vs. Impact Analysis

The size of a library is often a primary consideration when integrating it into an application. A larger library might be perceived as having a more significant impact on performance metrics, such as load time or bundle size. However, the actual impact of a library's size on performance can vary, and it's essential to analyze this relationship to make informed decisions. This section attempts to understand if there is a correlation between the size of each library and its corresponding impact on the application's performance when sharing them using Module Federation.

In order to optimize library sharing, the Dashboards Angular-Web-Component was selected for analysis. The initial examination focuses on identifying the larger bundles within the web component bundle prior to utilizing Module Federation. Before the implementation of shared libraries, the web component bundle had an estimated transfer size of 481.96 KB. An analysis conducted using the webpack-bundle-analyzer plugin revealed the libraries that most significantly impacted the final web component bundle size. These libraries are presented in Table 6.1.

Table 6.1: Libraries Raw Size results from webpack-bundle-analyzer for Angular-Web-Component dashboards.

Library	Raw Size (Kb)	Library	Raw Size (Kb)
@enterprise-ds/components	4510	@angular/platform-browser	95.15
@angular/core	1170	rxjs	52.73
@angular/cdk	371.51	ngx-device-detector	44.31
@angular/common	354.63	@ngx-translate/core	36.8
@angular/router	262.28	apollo3-cache-persist	25.5
@angular/forms	256.3	apollo-angular	22.31
@angular/animations	201.68	**ngx-image-compress**	21.21
sortablejs	113.83	-	-

Considering the data presented in the Table 6.1, some libraries, highlighted in bold, are not utilized in any other web components, consequently, sharing them using Module Federation would not be beneficial. Webpack sets a recommended threshold of 250 KB threshold for a single bundle (Webpack, 2023b). Taking all these libraries except the ones being only used by this Angular-Web-Component and adding them in a sequence manner results in the graph presented in Figure 6.1[1].

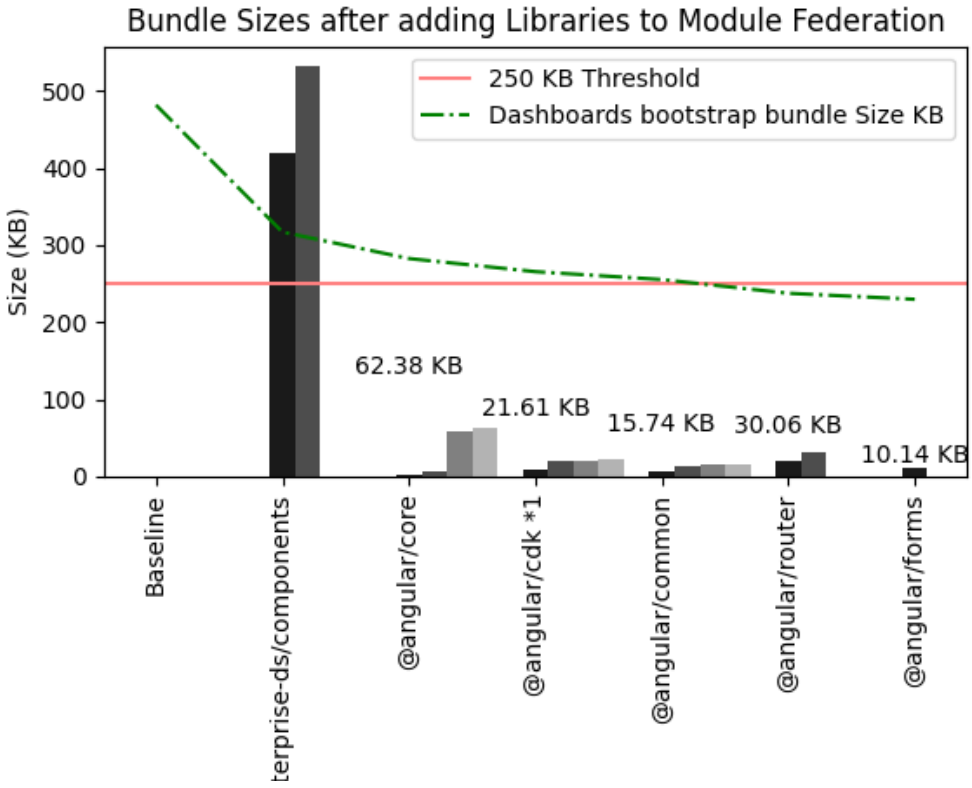


Figure 6.1: Bundle sizes after integrating libraries with Module Federation.

[1]For the @angular/cdk was only represented the top four bundles as this libraries generates multiple bundles that would not be able to be correctly represented in the graphic.

The graph depicted in Figure 6.1 illustrates that upon the inclusion of just four libraries (@enterprise-ds/components, @angular/core, @angular/cdk, and @angular/-common), the largest bundle size of the application falls below the recommended 250 KB threshold set by Webpack for a single bundle. The @enterprise-ds/components library generates two sizable bundles. While these bundles are loaded once and subsequently reused across various components, their substantial size may be caused from the inability to tree-shake components from this library. This limitation could be attributed to the absence of secondary entrypoints in the library, leading to the inclusion of all components regardless of their actual usage within the application. An update to this library to incorporate secondary entrypoints might align its behavior with that of @angular/cdk, which efficiently divides into multiple smaller bundles that can be lazy-loaded as required.

It's also observed that by sharing just six components, the application bundle size is reduced to 229.7 KB from an initial 481.96 KB. However, this reduction in bundle size also translates to an increase in network requests, the implications of which will be further discussed in chapter 6.4.3.

In order to understand the feasibility of sharing every library through Module Federation, a new test was conducted. This involved deploying versions of all web components (Angular-Web-Components and Widgets) without any shared libraries and subsequently adding one library at a time to the configuration of Module Federation in a cumulative way. Due to an issue happening at runtime, the libraries @angular/core and @angular/common were added together in a single test and therefore are the first ones to be tested. The process was then repeated for @enterprise-ds/components and other libraries. The outcomes, showcasing the

cumulative size of transferred JavaScript files post the integration of those libraries with Module Federation, are presented in Figure 6.2.

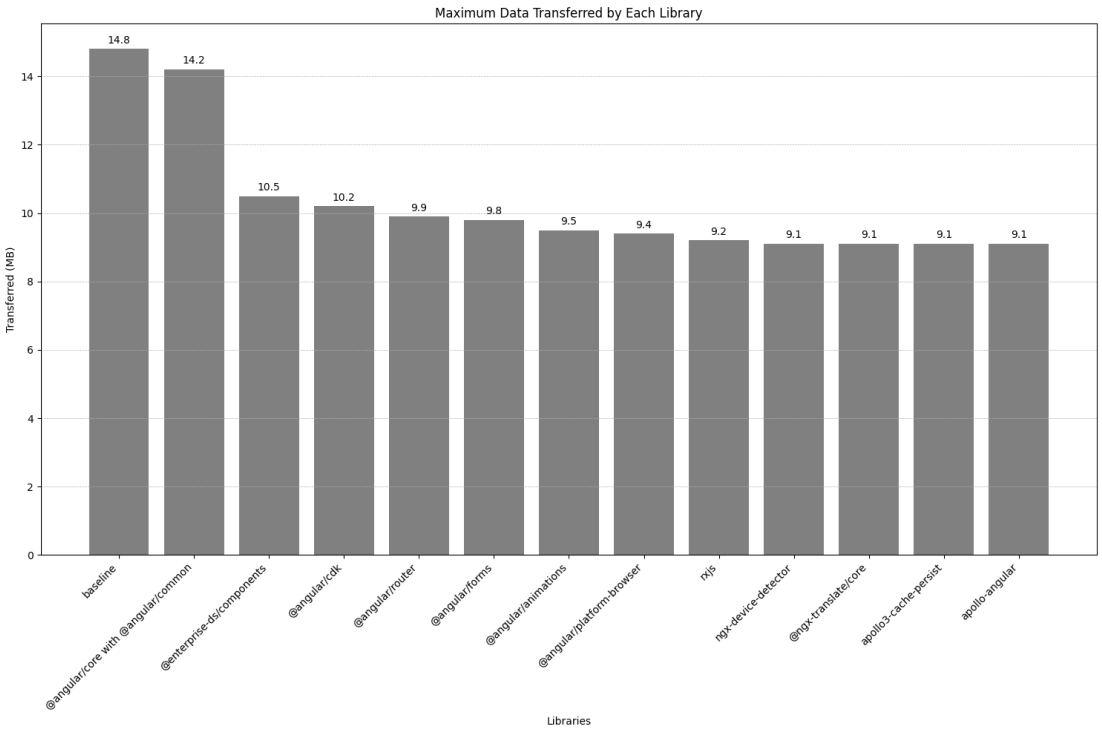


Figure 6.2: Graph bar of cumulative size of transferred JavaScript files after.

There was a notable reduction in the final bundle size of JavaScript files, decreasing from 14.8 MB to 9.1 MB. It was observed that after sharing the library ngx-device-detector, subsequent shared libraries had a minimal impact on the final bundle size of the application. These are the libraries that have a raw size below 50 KB as presented in Table 6.1.

To assess whether sharing libraries could enhance the application's loading speed, the load time was also measured and the graph presenting the measurements are illustrated in Fig 6.3.

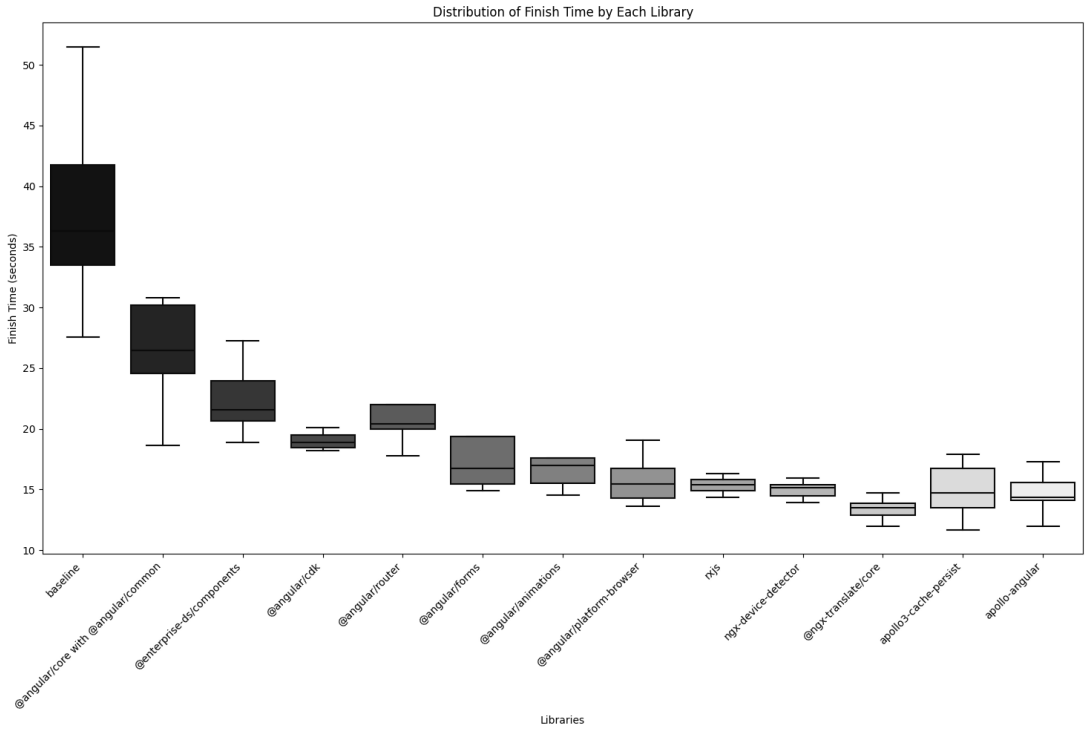


Figure 6.3: Box plot of cumulative size of transferred Javascript files.

From the analysis of the graph in Figure 6.3 it is observed a decrease in the loading time as more libraries are shared. However, the loading time starts to stay more consistent after adding @angular/forms library.

The load time, which represents the time taken for the application to load, exhibited a range of values with some big outliers. Outliers in data can significantly skew the mean, potentially leading to misleading interpretations, therefore the median, being the middle value of a sorted dataset, offers a more robust measure of central tendency as it remains unaffected by extreme values and provides a more representative snapshot of the typical load time. Given the presence of these outliers in the load time data, the decision was made to utilize the median as a more accurate

representation for the calculation of the Pearson Correlation Coefficient. This coefficient is a measure that quantifies the linear relationship between two datasets and was used to analyze the correlation between library size and transfer size and between library size and load time

For the relationship between library size and total transferred bundles size, the Pearson correlation coefficient was calculated to be 0.42. This indicates a moderate positive correlation between the library size and the transferred size. This suggests that as the size of a library increases, there is a tendency for the transferred size to decrease.

On the other hand, when analyzing the library size and median load Time, the correlation coefficient was found to be 0.61. This denotes a strong positive correlation between the library size and the load time. It implies that sharing larger libraries tend to have smaller load times, emphasizing the significant impact of library size on load performance.

6.4 Number of Applications vs. Impact Analysis

This sub-chapter evaluates the performance implications of integrating web components using Module Federation. The methodology employed for this assessment is both iterative and cumulative. Starting with a baseline code-base, a single web component was integrated using Module Federation. Subsequent to this integration, key performance metrics were measured and documented. Following this, another web component was added to the mix, and the same set of metrics were measured again. This process was repeated for each web component, ensuring that

every subsequent measurement captured not just the impact of the newly added component, but the cumulative effect of all previously integrated components as well. This entire evaluation was conducted within a controlled Test environment that mimics the Production environment. This environment was chosen to provide a consistent baseline for all measurements, ensuring that external variables were kept to a minimum. However, like in all controlled environments, variables such as internet speed fluctuations, server response times, or other environmental nuances could introduce variations in the results. As such, while the Test environment offers a stable platform for evaluation, results interpretation should take a degree of caution, acknowledging the potential for such external influences. Furthermore, to ensure consistency in the evaluation process and to isolate the impact of Module Federation as the primary variable, all tests and measurements were derived from the same foundational code-base. No other changes, modifications, or optimizations were introduced to the code during this evaluation. This approach was adopted to guarantee that any observed performance variations or anomalies could be confidently attributed to the cumulative integration of Module Federation.

6.4.1 Initial Load Time

The initial load time of a web application is a critical metric, as it directly impacts the user's first impression. With the integration of Module Federation, there might be changes in how components are loaded, which can influence this metric. Figure 6.4 depicts a box plot with the results of the initial load time per addition of web component.

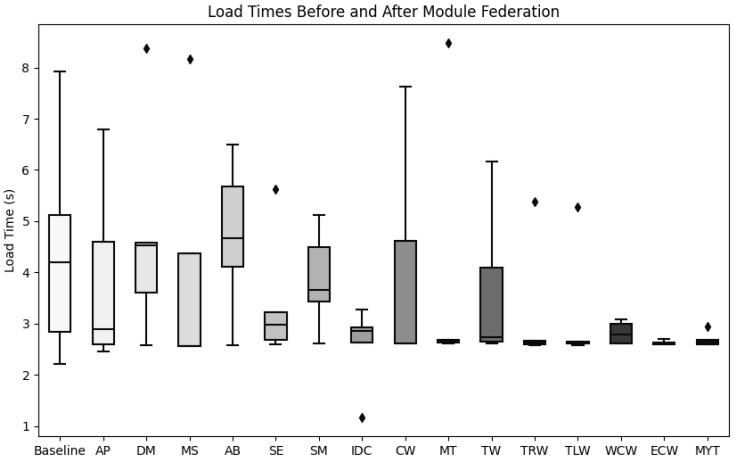


Figure 6.4: Box plot comparing load times before and after module federation integration.

In the box plot illustrated in Figure 6.4, the data does not immediately suggest a correlation between the integration of Module Federation and the initial load time. However in order to understand if there is a statistically significant difference in the initial load time made after integrating module federation compared to the baseline, a one-sample t-test was executed.

Hypothesis Testing

- H0: The integration of module federation does not significantly affect the initial load time.
- H1: The integration of module federation significantly affects the initial load time.

From the data, there appears to be a general reduction in load times after the integration of module federation, which aligns with the alternative hypothesis (H1). However, in order to validate this observation with a 95% confidence level,

a t-test was conducted. With a T-statistic value of 1.685 and a P-value of 0.095, which is greater than the threshold of 0.05, the analysis indicates that there isn't a statistically significant difference in load times before and after the incorporation of module federation.

6.4.2 Total Javascript Files Load Time

The time taken to load all JavaScript files can influence the interactivity and responsiveness of the application. With module federation, the way JavaScript files are bundled and loaded is different. This metric and the provided data represents the load time of Javascript files under different configurations of Module Federation. The initial set of values represents the load times without any module federation. Subsequent sets of values represent load times after introducing module federation to various components of the application.

Hypothesis Testing

- H0: The integration of module federation does not significantly affect the total JavaScript files load time.
- H1: The integration of module federation significantly affects the total JavaScript files load time.

In order to determine whether to reject the null hypothesis or not, a paired t-test will be used. It will be compared the mean load times before and after introducing Module Federation for each component in a cumulative manner. Figure 6.5 illustrates the box plot for the values read of total load times of Javascript files.

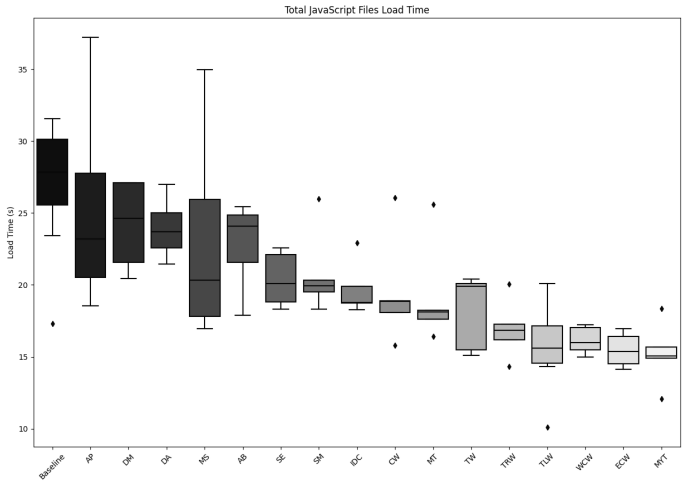


Figure 6.5: Box plot comparing total load times of Javascript files before and after module federation integration.

These results were also mapped to a line graph, represented in Figure 6.6 that represents the mean at each stage of the migration to module federation.

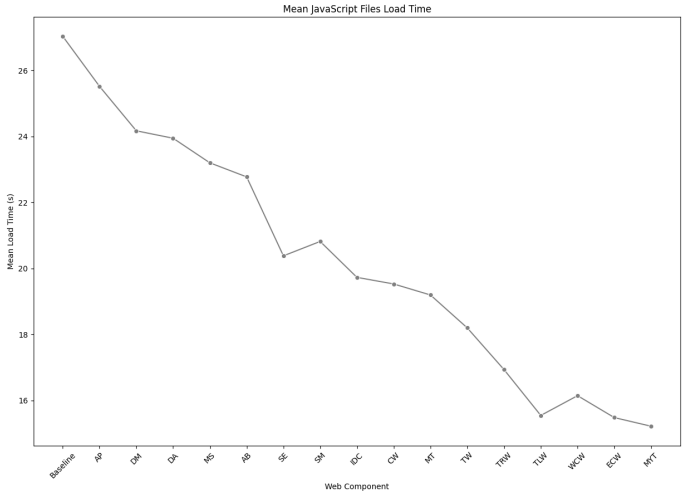


Figure 6.6: Line Graph comparing total load times of Javascript files before and after Module Federation integration.

The observed decrease in total load time suggests an improvement. To substantiate this observation with a 95% confidence level, a paired t-test was used. This test compared the baseline load times with those recorded after the integration of module federation into each component. Each data point in the Baseline set was systematically paired with a corresponding data point in the Module Federation set for every component. The detailed results are documented in Table 6.4. It was also calculated the percentage improvement in the loading of JavaScript files using the Equation 6.1.

$$\text{Percentage Improvement} = \frac{\text{Mean of Baseline} - \text{Mean of Component}}{\text{Mean of Baseline}} * 100 \quad (6.1)$$

Upon integrating all web components with module federation, there was a improvement of 43.70% in JavaScript file load times. The results of the paired t-test are presente in Table 6.4.

Table 6.4: Cumulative improvement of adding web components sharing dependencies.

Component	Improvement	P-value	Component	Improvement	P-value
AP	5.61%	0.55	CW	27.75%	0.17
DM	10.59%	0.54	MT	28.99%	0.07
DA	11.42%	0.51	TW	32.67%	**0.05**
MS	14.18%	0.56	TRW	37.35%	**0.02**
AB	15.75%	0.22	TLW	42.49%	**0.01**
SE	24.60%	**0.05**	WCW	40.27%	**0.01**
SM	22.99%	0.07	ECW	42.73%	**0.01**
IDC	27.01%	**0.04**	MYT	43.70%	**0.01**

As highlighted in the table, by the inclusion of the 10th web component, the P-value is consistent less than or equal to 0.05. This result leads to the rejection of the null hypothesis. It signifies that the improvement in load times, post the integration of Module Federation for a specific number of components, is statistically significant.

6.4.3 Number of Requests

The number of requests made by a web application can influence its performance, especially on slower networks. With module federation, components are loaded differently, potentially changing the number of requests. The results of the number of requests made by the different web components compared with the baseline is presented in the Figure 6.7.

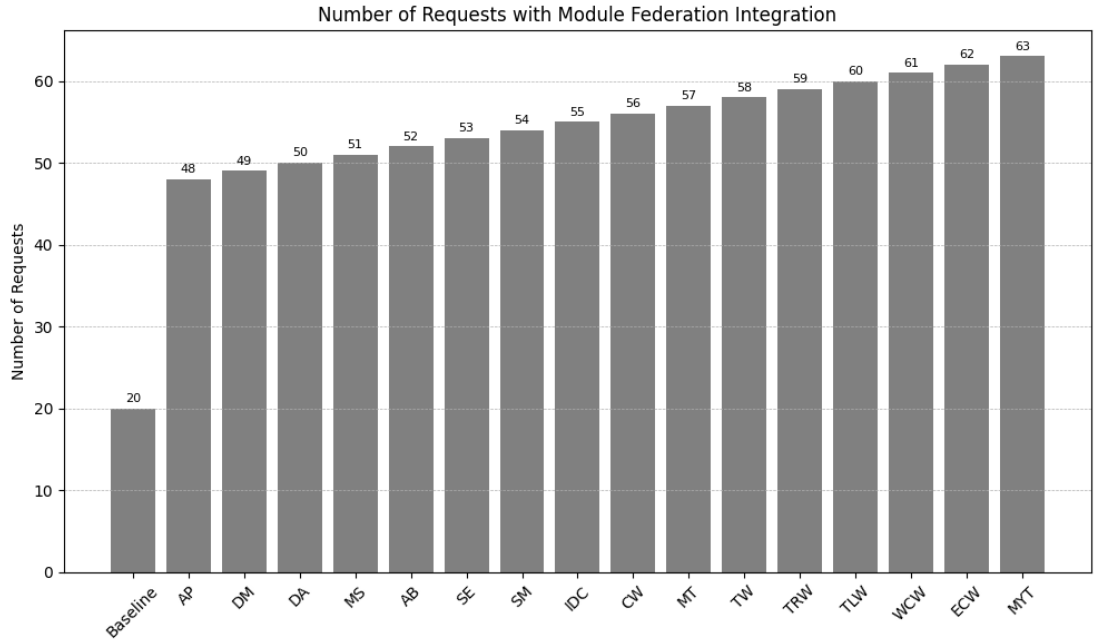


Figure 6.7: Cumulative number of requests made after adding web components using Module Federation.

The graphic in Figure 6.7, presents that as more components are integrated with module federation, the number of requests consistently increases. The initial jump in requests when the first module federation is added to the first component is noticeable, and subsequent additions of module federation to other components result in incremental increases in the number of requests. This pattern suggests that each integration of module federation contributes to the overall increase in requests. The one-sample t-test was conducted to determine if there was a statistically significant difference in the number of requests made after integrating module federation compared to the baseline.

Hypothesis Testing

- H0: The mean number of requests after integrating module federation is equal to the number of requests without module federation.
- H1: The mean number of requests after integrating module federation is different from the number of requests without module federation.

The t-statistic value obtained was 29.83, and the associated p-value was close to zero. Given a significance level of 0.05, the p-value is considerably smaller. This leads to the rejection of the null hypothesis. The results indicate that there is a statistically significant difference in the number of requests after integrating module federation. Specifically, the number of requests made after the integration of the first web component using Module Federation is significantly higher than the baseline.

6.4.4 Total Data Transferred in MB (JS files)

The total data transferred, especially for JavaScript files, can impact load times and user experience, especially on limited or metered connections. The measured data transfer size zis presented in Figure 6.8.

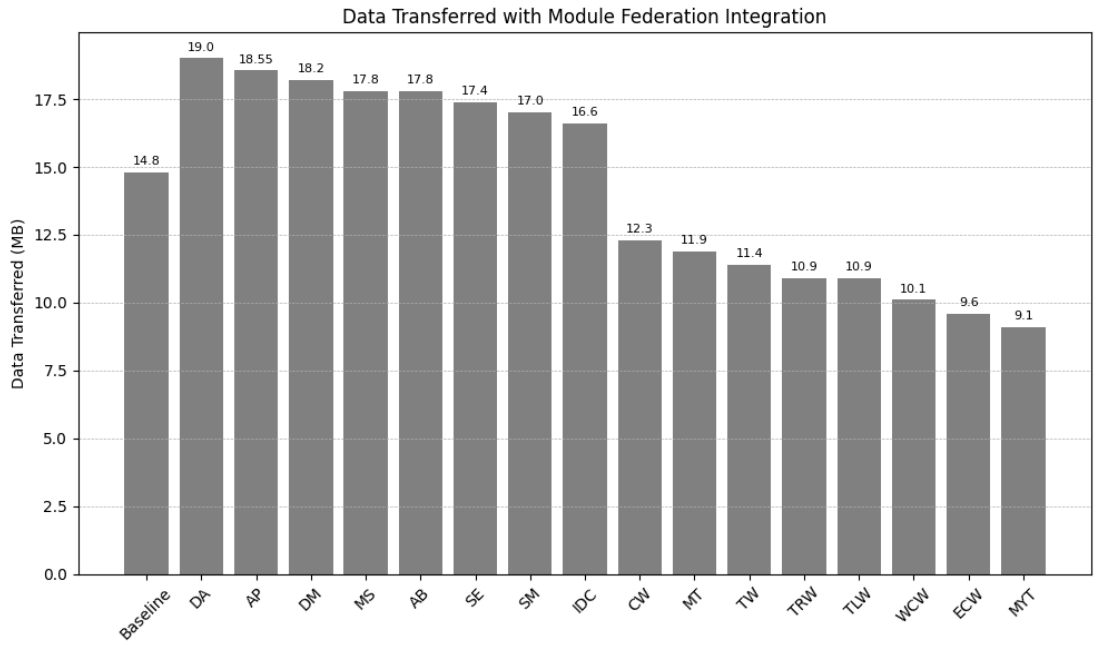


Figure 6.8: Data Transferred with Module Federation Integration

Without the use of Module Federation the data transfer size 14.8 MB. As each component is integrated with Module Federation, there is a consistent reduction in the data transferred, after all integrations with Module Federation, the data transfer has reduced to 9.1 MB. This trend suggests that module federation is effective in reducing the amount of data transferred, which can lead to faster load times and a more efficient application.

The integration of Module Federation into the application has presented impact on

its performance metrics. Starting with the total data transferred in MB (Javascript files), there was a noticeable reduction in the amount of data transferred as more components used Module Federation. This reduction suggests a more efficient code-sharing mechanism and potentially faster load times due to fewer data being fetched. However, when observing the number of requests, there was a consistent increase as more components were integrated with Module Federation. This rise in requests could introduce potential network overhead, especially in unstable network conditions.

In terms of the total Javascript Files load time, the application experienced a significant improvement, which directly contributes to a better user experience. Lastly, the initial load time of the application, which is a critical metric for user engagement, showed varying results. Overall the integration of the Module Federation into the web components led to less initial load times

6.5 Summary

This chapter assessed the effects of Module Federation on application performance. Through experiments, the relationship between shared dependencies, bundle sizes, and loading speeds was explored. Metrics were analyzed, and statistical tools were used to validate findings. The chapter provided insights into the practical benefits and challenges of implementing module federation.

Chapter 7

Conclusions

This chapter presents the conclusions from the study on the benefits and challenges of using module federation. It also provides recommendations and outlines the next steps.

7.1 Performance and Bundle Size

One of the most significant findings from this study is the advantage Module Federation offers in reducing application bundle sizes. A smaller bundle size translates to quicker load times, enhancing the user experience, however, these benefits only occur when multiple applications share dependencies. This implies that for organizations with a suite of interconnected applications, Module Federation can be used for enhancing applications load time. As the number of shared libraries increases, the benefit in terms of application loading speed diminishes for smaller libraries. While the loading speed might not show a noticeable difference, the

overhead of additional requests becomes evident. This is an essential consideration for application's owners, as the balance between the number of shared libraries and the associated overhead needs careful calibration. Figure 7.1 illustrates a network print screen after the integration of Module Federation in the application.

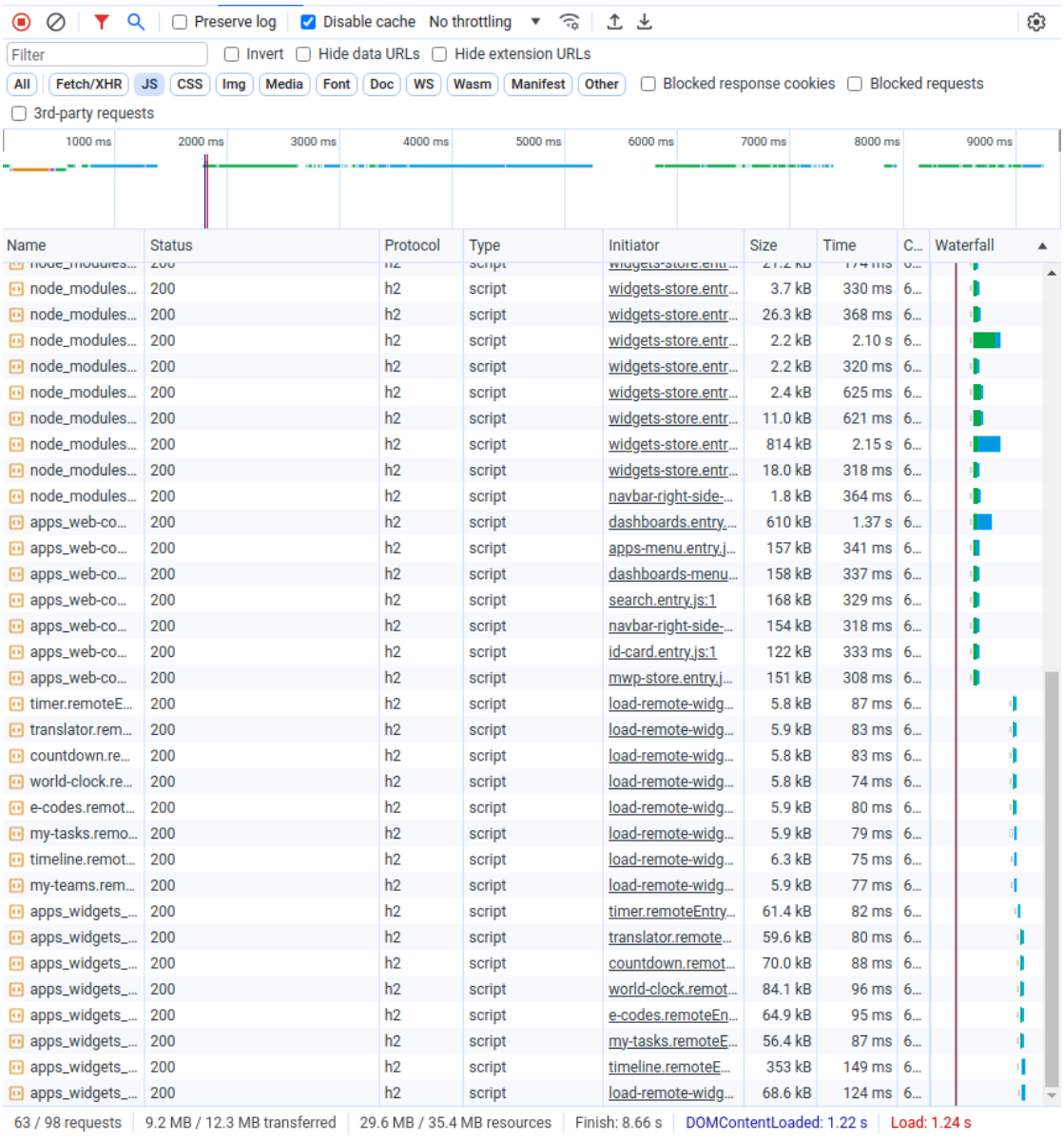


Figure 7.1: Network print screen after adding Module Federation.

One of the technical leaders of the team added that the implementation and experience with "Module Federation minimizes the memory footprint of our micro-frontends and drastically improves the performance of our application integration platform, which is crucial for our users to have a joyful and efficient experience".

In conclusion, Module Federation offers some benefits in terms of code efficiency and certain aspects of load performance, it also introduces challenges in the form of increased requests. Even with a increasing number of requests, the load times were being reduced after the use of Module Federation, potentially meaning that the use of this technology would only make sense to project applications that make a heavy use of Module Federation with the use of multiple federated components.

7.2 Implementation and Team Collaboration

From an implementation perspective, module federation stands out for its simplicity and comprehensibility. Developers can integrate it into their applications with relative ease, making it an attractive option for teams looking to optimize their applications without a big learning curve.

However, the introduction of module federation does necessitate a shift in team dynamics and collaboration. When different teams manage various parts of an interconnected application ecosystem, a consensus on shared dependencies becomes crucial. This is especially true for Angular related dependencies, given the project stipulation to maintain at least the same major version across shared dependencies. Such collaboration ensures consistency and prevents potential conflicts or bugs arising from version mismatches.

7.3 Future Recommendations

While module federation offers numerous advantages, it's essential to keep an eye on emerging technologies and standards. One such recommendation is the adoption of `import maps`. Being a native solution now supported by all major browsers, including those used by the company, import maps present a promising alternative. Their native nature ensures optimal performance and compatibility, making them a worthy successor to the current implementation without being bonded to any compiler tool.

Other recommendation is to create either a Command-line Interface (CLI) tool or a angular generator to create widgets supporting Module Federation, so external teams can create their federated applications without the need to spend time understanding the processes behind Module Federation.

In conclusion, Module Federation emerges as a easy to use tool for modern web development, offering benefits in performance optimization and total bundle size reduction. However, like all tools, its effective utilization requires a understanding of its strengths, limitations, and the broader ecosystem in which it operates. With careful implementation and collaboration, module federation can significantly elevate the performance and efficiency of web applications.

References

Angular. (2022). *What is angular?* Retrieved from https://angular.io/guide/what-is-angular

Bevacqua, N. (2020). *Mastering modular javascript* (O'Reilly, Ed.). O'Reilly.

browserify. (2023). *Browserify* [Webpage]. Retrieved from https://browserify.org/

Cannavacciuolo, C., & Mariani, L. (2022, February). *Smoke testing of cloud systems.* https://doi.org/10.1109/ICST53961.2022.00016

Chauhan, V. K. (2014, February). *Smoke testing. 4.* Retrieved from https://api.semanticscholar.org/CorpusID:198952328

Clark, L. (2018). *ES modules: A cartoon deep-dive* [Webpage]. online.

Concept. (2023). Retrieved from https://webpack.js.org/concepts/

Critical TechWorks. (2023). *We are changing the way the world moves.* Retrieved from https://www.criticaltechworks.com/

devtools-fm. (2022). Zack jackson - module federation. Retrieved from https://www.youtube.com/watch?v=XpeD4FtlMg4

Ebey, J. (n.d.). *Module federation.* Retrieved from https://module-federation.github.io/

Farrell, B. (2019). *Web components in action.* Manning Publications Co.

Fowler, M. (2001). Reducing coupling. *IEEE Software*, *01*(0740-7459).

Fowler, M. (2004). *StranglerFigApplication.* Retrieved from https://martinfowler.com/bliki/StranglerFigApplication.html

Ghadyani, K. (2021). Webpack module federation in-depth w/ zach jackson #codeconversation. Retrieved from https://www.youtube.com/watch?v=d1SS7KAsbdY&t=205s

Grini, H. (2021). *Micro frontends with webpack module federation.* Retrieved from https://www.teliacompany.com/en/about-the-company

Herbold, S., & Haar, T. (2022). Smoke testing for machine learning: Simple tests to discover severe bugs. *Empirical Software Engineering, 27.* https://doi.org/10.1007/s10664-021-10073-7

Herrington, J., & Jackson, Z. (2023). *Practical module federation* (2nd ed.). ScriptedAlchemy.

Housing. (2023). *Welcome to housing.* Retrieved from https://housing.com/about

Jackson, Z. (2020). *Streaming code and payloads in multi-threaded, parallel servers with module federation.* Retrieved from https://www.youtube.com/watch?v=kOuoSBTCzl4

Jackson, Z. (2021). *Module federation, how do we create unit tests for distributed code?!* Retrieved from https://scriptedalchemy.medium.com/module-federation-how-do-we-create-unit-tests-for-it-bd0d73c999bc

Jackson, Z. (2021). *Next.js 11, module federation, and ssr - a brave new world.* Retrieved from https://javascript.plainenglish.io/next-js-11-module-federation-and-ssr-a-whole-new-world-6da7641a25b4

Jackson, Z. (2022). *When should you leverage module federation, and how?* [Webpage]. Retrieved from https://scriptedalchemy.medium.com/when-should-you-

leverage-module-federation-and-how-2998b132c840

Jackson, Z. (2023a). *Module federation examples*. Retrieved from https://github.com/module-federation/module-federation-examples

Jackson, Z. (2023b). Module federation: The federated applications revolution. Retrieved from https://www.infoq.com/presentations/module-federation/

Jartarghar, H. A., Salanke, G. R., R, A. K. A., S, S. G., & Dalali, S. (2022). React apps with server-side rendering: Next.js. Retrieved from https://jtec.utem.edu.my/jtec/article/view/6192/4083

Koppers, T. (2023). *Code splitting*. Retrieved from https://webpack.js.org/guides/code-splitting/

Latendresse, J., Mujahid, S., Costa, D., & Shihab, E. (2022). *Not all dependencies are equal: An empirical study on production dependencies in npm*. https://doi.org/10.48550/arXiv.2207.14711

LeSS. (2023). *Teams* [Webpage]. Retrieved from https://less.works/less/structure/teams

Maida, K. (2017). Migrating an angularjs app to angular. *Auth0*. Retrieved from https://assets.ctfassets.net/2ntc334xpx65/4I8HgPbZjq4GMMi2SM4mmG/8478dd9123d80a91ee2c997ed5c1befa/migrating-to-angular.pdf

McClamrock, R. (2006). *Modularity*. https://doi.org/10.1002/0470018860.s00168

MDN contributors. (2023a). *Import()*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import

MDN contributors. (2023b). *JavaScript modules*. Retrieved from https://develope

r.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

Mezzalira, L. (2019). *Micro-frontends decisions framework*. Retrieved from https://lucamezzalira.medium.com/micro-frontends-decisions-framework-ebcd22256513

Microsoft. (2023). *Strangler fig pattern*. Retrieved from https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig

Mohan, M., & Prusty, N. (2018). *Learn ecmascript - second edition*. Packt Publishing.

Mozilla. (2023a). *IIFE* [Webpage]. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Web_components

Mozilla. (2023b). *IIFE* [Webpage]. Retrieved from https://developer.mozilla.org/en-US/docs/Glossary/IIFE

Mozilla. (2023c). *Import maps*. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script/type/importmap

Nishizu, Y., & Kamina, T. (2022). Implementing micro frontends using signal-based web components. *Journal of Information Processing*, *30*, 505–512. https://doi.org/10.2197/ipsjjip.30.505

Node.js. (2023). *Node.js v19.6.0 documentation* [Webpage]. Retrieved from https://nodejs.org/api/modules.html

npm. (2021). *About semantic versioning*. Retrieved from https://docs.npmjs.com/about-semantic-versioning

NPM Trends. (2023). *Esbuild vs parcel vs rollup vs webpack*. Retrieved from

https://npmtrends.com/esbuild-vs-parcel-vs-rollup-vs-webpack

Nrwl. (2023). *Share your cache*. Retrieved from https://nx.dev/core-features/share-your-cache

Parcel. (2021). *Announcing parcel v2!* Retrieved from https://parceljs.org/blog/v2/

Parcel. (2023). *Parcel*. Retrieved from https://parceljs.org/

Polylith. (2022). *Polylith in a nutshell* [Webpage]. Retrieved from https://polylith.gitbook.io/polylith/introduction/polylith-in-a-nutshell

Possumato, M., Tomlin, N., Andree, J., Shim, A., & Pilani, R. (2021). *How we build micro frontends with lattice*. Retrieved from https://netflixtechblog.com/how-we-build-micro-frontends-with-lattice-22b8635f77ea

Preston-Werner, T. (n.d.). *Semantic versioning 2.0.0*. Retrieved from https://semver.org/

Rauschmayer, A. (2022). *JavaScript for impatient programmers ecmascript 2022 edition*. exploringjs.com.

Richardson, C. (2023). *Pattern: Strangler application*. Retrieved from https://microservices.io/patterns/refactoring/strangler-application.html

Rivian. (2023). *Our company*. Retrieved from https://rivian.com/our-company

RollupJs. (2023). *Introduction*. Retrieved from https://rollupjs.org/introduction/

Saini, N. (2023). *Module federation pipeline — part 1*. Retrieved from https://medium.com/engineering-housing/module-federation-pipeline-part-1-6c81ea15fe16

Silva, R. A. P. da. (2021). *A micro frontends solution - analyzing quality attributes*. ISEP - Instituto Superior de Engenharia do Porto.

Single-Spa. (2023a). *Concept: Microfrontends*. Retrieved from https://single-spa.js.org/docs/microfrontends-concept

Single-Spa. (2023b). *Frequently asked questions*. Retrieved from https://single-spa.js.org/docs/faq/

Single-Spa. (2023c). *The recommended setup*. Retrieved from https://single-spa.js.org/docs/recommended-setup

Steyer, M. (2022a). *Announcing native federation 1.0*. Retrieved from https://www.angulararchitects.io/en/blog/announcing-native-federation-1-0/

Steyer, M. (2022b). *Import maps: The next evolution step for micro frontends*. Retrieved from https://www.angulararchitects.io/en/blog/import-maps-the-next-evolution-step-for-micro-frontends-article/

Steyer, M. (2023). *Beyond micro frontends - three additional things module federation makes possible*. NG-DE Conference; video.

Taibi, D., & Mezzalira, L. (2022). Micro-frontends: Principles, implementations, and pitfalls. *ACM SIGSOFT Software Engineering Notes*, *47*, 25–29. https://doi.org/10.1145/3561846.3561853

Telia. (2023). *We are telia*. Retrieved from https://www.teliacompany.com/en/about-the-company

Thompson, M. (2023). *Discontinued Long Term Support for AngularJS* [Blog]. Retrieved from https://blog.angular.io/discontinued-long-term-support-for-

angularjs-cc066b82e65a

Unbounce. (n.d.). *The ultimate guide to a/b testing.* Retrieved from http://www.datascienceassn.org/sites/default/files/A-B%20Testing%20Guide.pdf

Wallace, E. (2023). *Why is esbuild fast?* Retrieved from https://esbuild.github.io/faq/

Walsh, D. (2019). *How to design and analyze online a/b tests within decentralized organizations.* Retrieved from http://purl.stanford.edu/yv309nh2575

Webpack. (2023a). *Module federation* [Webpage]. Retrieved from https://webpack.js.org/concepts/module-federation/

Webpack. (2023b). *Performance* [Webpage]. Retrieved from https://webpack.js.org/configuration/performance/

Zaikin, V. (2023). *You might not need module federation: Orchestrate your microfrontends at runtime with import maps.* Retrieved from https://www.mercedes-benz.io/2023/01/05/you-might-not-need-module-federation-orchestrate-your-microfrontends-at-runtime-with-import-maps/

Attachment I

Table 7.1: Mapping of web components acronyms with full names.

Acronym	Full Name
AP	Apps Menu
DM	Dashboards Menu
DA	Dashboards
MS	MWP-Store
AB	Announcements Banner
SE	Search
SM	Settings Menu
IDC	ID Card

Acronym	Full Name
CW	Countdown Widget
MT	My Teams Widget
TW	Timeline Widget
TRW	Timer Widget
TLW	Translator Widget
WCW	World Clock Widget
ECW	E-Codes Widget
MYT	My Tasks Widget