HKA

Karlsruhe University of Applied Sciences

Department of Computer Science and Business Information Systems
Business Information Systems

MASTER´S THESIS

Evaluating methods of avoiding redundant libraries in a mirco-frontend based landscape

Author	Mr. Viktor Sperling
Matriculation number	71197
Workplace	SAP Hybris GmbH, München
First Advisor	Prof. Dr. Udo Müller
Second Advisor	Prof. Dr. Rainer Neumann
Closing Date	31 March, 2022

31 March, 2022

Chairman of the Examination Board

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die hier vorgelegte Master-Thesis selbstständig und ausschließlich unter Verwendung der angegebenen Literatur und sonstigen Hilfsmittel verfasst habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

\line

Karlsruhe, den 21. März 2022

Abstract

This transcript provides proof of possible solutions regarding redundant libraries in micro frontend landscapes. The data collected is based on an implemented prototype using the Luigi micro frontend framework. Different landscapes, with different technologies are implemented to represent a heterogenic landscape, which is considered to be representative for a micro frontend landscape.

The technologies in use are a Content Delivery Network, the Compound feature of Luigi in combination with Web Components and lastly the Webpack Module Federation, a rather new feature of Webpack which can serve as a foundation for micro frontend architectures. After implementation, data is collected and displayed, defining the performance metrics used for the evaluation. It is proven that every of the respective technologies has their own means to avoid redundant libraries in the micro frontend landscapes. Let it either be distinguishable resource URLs provided by the CDN, reusable Web Components or even shared dependency through the Module Federation. Each technology shows ways of solving the issue at hand. Still every method has its limits and these are evaluated as well, such as the effort involved when using one of the corresponding technologies.

For CDN it is shown, that even though easy to implement and use in the project itself, a public CDN has certain restrictions. This include the absence of special libraries required for the project or the constant dependency on the CDN servers to be available. Hosting an own CDN involves further costs and requires time for implementation and configuration, which in certain cases might not be affordable. Web Components implemented as Compounds in Luigi, show great potential in avoiding redundancies to a certain degree, by providing reusable components registered in the browser. Additionally this technology is framework independent and compatible with almost all modern browsers, due to its standard status. It is shown how this technology provides a way to support multi-version landscapes via the scoping feature of the UI5 Web Components library. The Webpack Module Federation, is considered to be a powerful tool to solve the issue of the redundancies in micro frontend landscapes. Not only can it serve to share central dependencies between its federalized modules, but also optimize shared dependencies in different versions. It is assumed that more features can be enabled through the Webpack bundler to optimize the landscape even further.

After showing the means on how each respective technology avoids redundant libraries in a micro frontend landscape, the comparison is made and a conclusion is drawn. The use case and the requirements on the landscape define which technology should or could be used, since every technology provides a solution for the issue in their own way.

Contents

1. Introduction

This chapter will explain the problem, motivation and goals of this transcript. It will also narrow down the thesis scope.

1.1 Motivation

The micro frontend technology has experienced a rise in popularity over the past 3 years. The occurrence of such a trend is no surprise, considering the fact that more and more applications are designed following the micro service architecture principles.[38] Micro frontends share a common interest with their assumed precursor. Generally it can be stated, that the market prefers smaller independent lightweight applications over monolithic giants with strong dependencies. Similar to micro services, micro frontends offer almost the same advantages to a development team: Independence of the respective micro frontend teams, shorter and decoupled developments cycles, smaller scope of maintenance, separations of concerns, reusable components available via defined APIs and further more.[46] But still this concept has certain flaws which have to be considered when making the decision.[45]

Depending on the type of the to-be-developed application, the concept of micro frontends is not always favored. One major point of criticism is the aspect of redundancies inside the landscape. Specifically the aspect of isolation is a core aspect of this architecture. The circumstance that, each micro frontend is a separate project with its own runtime, developed by an independent team causes the issue to occur. It allows the development teams to use their own preferred tech stack and enables the application itself to be reused in different landscapes. Nonetheless, that also means that isolated micro frontends are partially loading and using the same libraries during their runtime. This is causing an overhead on client and server side and therefore correlates with the runtime costs of such a landscape and potentially an inferior user experience.[28][12]

Despite isolation being a key aspect of this architecture, it is still possible to avoid the caused redundancies without disabling it.

1.2 Goals

The goal of this thesis is to provide proof and evaluate methods to avoid redundancies during the runtime of a representative micro frontend landscape.

For the context of this transcript, the term *"redundancies"* or *"redundant libraries"* means same libraries or dependencies used by multiple micro frontend projects. It does not mean the redundant code elements, like reoccurring methods inside the projects or same CCS classes.

A library is determined by its identifier or its name, e.g. `@luigi-project`. If the given library or dependency is imported, via any means and in any version, by multiple different micro frontends, it is considered to be redundant for the context of this thesis. The case of different versions of redundant libraries is considered too and is covered in the following chapters.

To achieve the mentioned goal, two prototypes were implemented using the Luigi micro frontend framework. The landscape itself was implemented to be as representative as possible by using a heterogenic tech stack. Additionally following methods were implemented to avoid redundantly loaded libraries:

- Content Delivery Network (CDN)

- Web Components in Luigi Compound views

- Webpack Module Federation (WMF) framework

The implemented methods are then evaluated under predefined metrics. The steps followed to collect the data for the metrics are explained too. Eventually a conclusion is made based on the empiric data, accompanied by the subjective experience of the author. It is explained how the usage and implementation of the respective technologies was experienced and which advantages each technology brings to the table from an objective and subjective view.

1.3   Scope of Work

Since the time given for this transcript is limited, the context is scoped. Therefore, following aspects are either not considered or referenced on a theoretical level.

- The development of an own CDN - An approximate cost assumption is made for such a project, without actual implementation.

- Usage of the Webpack bundler outside the context of the Module Federation - Since this bundler offers many ways for configuration, the focus lies on the necessary onces for the WMF context.

- Configuration for the Luigi framework of the implemented landscape - Each prototype implemented uses this framework for their micro frontend landscapes and since not all configurations can be shown here, only a general overview is given.

- Development of Web Components - The general usage and functionality of this standard is explained.

- Combinations of the implemented technologies - This aspect is not empirically considered in this transcript.

1.4   The collaborators

This thesis was written in collaboration with SAP Luigi project team. The SAP originally founded in in 1972 as SAPD (system analysis program development), later became SAP AG in 2005 and in 2014 SAP SE. The Hybris company originally founded in 1997 was later acquired by the SAP in 2013, after moving its headquarter from Zug in Switzerland to Munich in Germany. The Luigi project team is part of the SAP Hybris organization but the developed framework is an open-source product. The team and their product aim to improve the experiences of customers, developers and administrators who are using Luigi. Their open-source product was designed to make the transformation from monolithic architectures into micro frontend based landscapes as smooth as possible.

2. State of technology

In this chapter the current state of technologies which can provide possible solutions for the issue of redundant resources in a micro frontend landscape is explained. Additionally, an overview about the used Luigi framework is given.

2.1 Micro frontend framework Luigi

Currently there are different micro frontend frameworks present on the market. Some of which are **Bit, SystemJS, Webpack 5 Module Federation, Piral, Single SPA** and **Luigi**.[41] For the implementation of the representative landscape the Luigi framework was used, therefore a short introduction of it is given here.

2.1.1 Luigi

Luigi is an open source JavaScript framework for micro frontends, consisting of two main parts, the **Luigi Core** and the **Luigi Client**. It provides a basis to integrate micro frontends, also called **Nodes** in this context. Both parts serve different purposes in the micro frontend landscape.[26]

2.1.1.1 Core

The **Luigi Core** is the groundstone of the whole landscape. It defines the main app, which will serve as an entry point for the user. The possible, applicable configurations of this part are the following:

- Navigation
- Authorization
- Localization - providing translations to display its applications in multiple languages
- General settings - e.g. header display configurations, apply loading indicators for the micro frontends, etc.
- Luigi Core API - providing functions for the core app to interact with the framework and access its features

The core application inside the landscape is determined via the `luigi.config.js`. Any project containing it can fulfill this role, which is the only prerequisite. In the context of the framework this app is later referred to as the **Core**. Following this principle, the project structure of the **Core** could look as shown in 2.1.[1]

```
1   - react-core-mf
2     - [...]
3     - node_modules
4     - public
5       - [...]
6       - index.html
7       - luigi.config.js
8     - [...]
9     - src
```

LISTING 2.1: Project structure for a Luigi Core application including the `luigi.config.js`

In the `luigi.config.js` itself the above mentioned possible configurations are defined.[25]

2.1.1.2 Client

The Luigi Client serves as the connection between the framework and its micro frontends. In order to establish the connection, a micro frontend has to import and initialize it. This will grant the micro frontend access to the **LuigiClient** object during the runtime. The micro frontend can then interact with the framework to e.g. set a global state, add event listeners or enable navigation between other micro frontends in the landscape.

An import of the Luigi Client can be accomplished via different methods. The most forward approach might be the direct import with a HTML script tag. Another option can be the import of the local package manager dependency.[24]

```
1   <!-- Via the HTML script tag -->
2   <script src="https://unpkg.com/@luigi-project/client@1.17.0/
     luigi-client.js">
3   </script>
4
5   <!-- Via the package manager -->
6   import LuigiClient from "@luigi-project/client";
```

LISTING 2.2: Import methods of the Luigi Client

---

[1]This example is directly taken from the implemented core project.

2.1.2 Architecture

After the short introduction of the framework's core components, a general architecture of a landscape implemented with it is provided below.
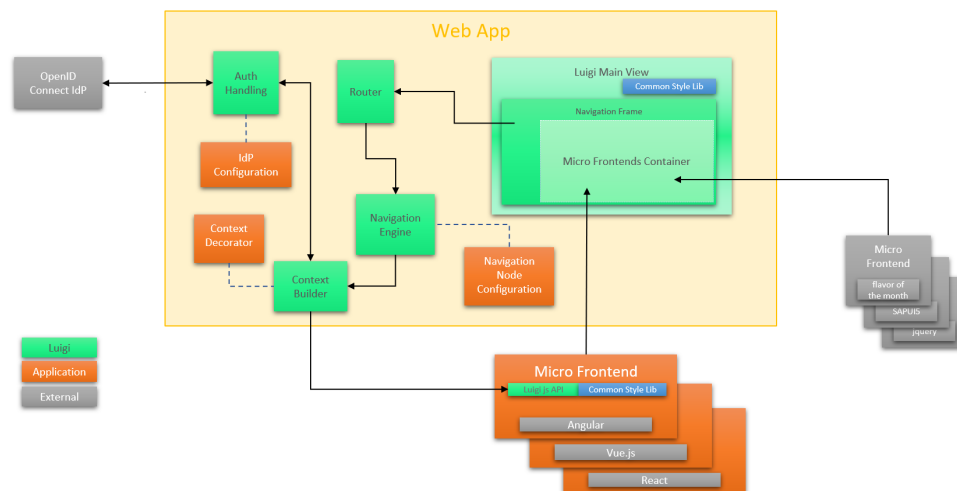

FIGURE 2.1: Architecture of the Luigi Framework [23]

As can be seen in figure 2.1, the displayed micro frontends in the Web App have a distinguished position in the Micro Frontends Container. The embedded projects can either be integrated via iFrames or as Web Components. Via the provided dashboard, called Navigation Frame in the image, the Luigi Core features are used. Through it the navigation or search feature can be accessed.

As described in the first chapter of this transcript, the issue in such a landscape is redundancies between the same libraries of the embedded micro frontends. Each Node is an isolated application, which is the reason for this problem. For instance, navigating to the first micro frontend and then followed by the second one will load the bundled projects with their corresponding dependencies. In this case the browser has no way to distinguish if a loaded resource is already present or not. Reason for that is the bundling of the projects which makes the references indistinguishable to the browser. The sections below will introduce methods of resolving exactly that issue.

2.2 Content Delivery Networks

One way of avoiding loading redundant libraries into a micro frontend landscape can be a central source, where all resources are stored and can be requested via an API. Such a

concept exists in the idea of Content Delivery Networks (CDNs). A CDN is the simplest way of solving the issue at hand. Since the resources are loaded via the CDN API, a dependency is available under a unique URL. Loading a dependency twice means the browser requests the resource from the same URL with the same parameters. Therefore, the browser can determine if the given resource is already present and decide to either load it from the CDN or from the browser cache, if it is already loaded.

Nonetheless, a CDN has its limits which have to be considered. Costs can play a key role in the decision process. Even though free CDN services like **Unpkg.com** or **cdnjs** are valid options, the advantages of using such might differ depending on the business case. For instance, those services might solve the issue of redundant libraries in a micro frontend landscape, but do they really improve the user experience of the application? Since the resources would be loaded from a cloud based service, the server location has significant influence on the latency of the requests. Therefore, depending from where the website is accessed, this circumstance could lead to a performance decrease when initially loading the page.

Also the technological dependency on a public cloud based CDN has to be considered. Most cloud based services are scalable and replicable. However, a down time could lead to immense costs for a platform provider relying on a public CDN to deliver the platform resources. Besides, it is never guaranteed that certain dependencies are always available on the public CDN.

Another option is self-hosting a CDN and providing all required resources independently. A business would have to acquire hardware-resources (physically or via a hardware-as-a-service provider like Amazon Web Services), maintain the servers, fill the CDN with the necessary content and keep it up to date. The costs attached to this scenario can be so immense and exclude this method due to cost inefficiency.[39]

2.3 Web Components

Another option to address the issue of redundancies are Web Components. Consisting of the following four standards, they provide reusable elements closed in HTML tags.[10]

- Custom Elements
- Shadow DOM
- HTML Template
- ES Modules

The necessary feature to achieve the goal of avoiding the issue is mainly provided by the **Custom Elements** standard of Web Components, therefore a short explanation of its behavior is given here.

2.3.1 Custom Elements

This standard of Web Components provides an API via which new HTML tags can be defined and registered by the `CustomElementRegistry`. Since one tag name can only be registered once, multiple registrations of the same element would lead to an exception. Thus, making already registered tags reusable for the whole micro frontend landscape without reloading the code of the registered element.[3]
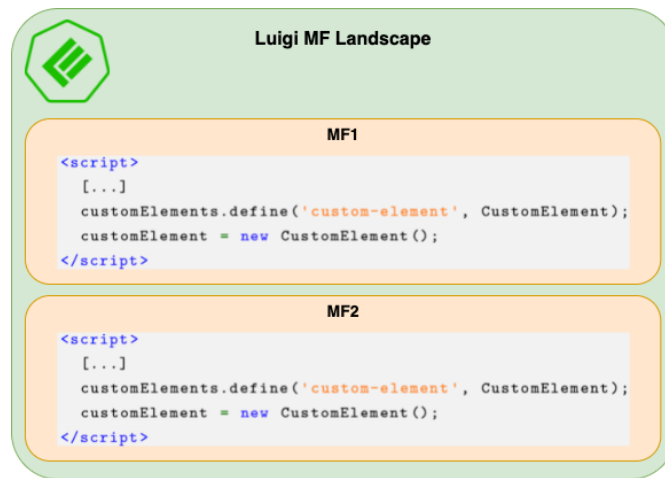
FIGURE 2.2: Simple micro frontend landscape using Web Components

Figure 2.2 shows a simple micro frontend landscape with Luigi using Web Components. As it is visible, both micro frontends are registering the same tag name `custom-element` using the `customElements` object. This object is a read-only property of the `Window` interface. The above shown scenario will cause a `DOMException`, this way making it impossible to register the same tag twice.

The registration of Web Components follows the *"first come, first served"* principle, the first micro frontend to register a tag name defines this tag and can not be overwritten without reloading the page.[8]

2.3.2 Objections

There is still a niche case which can be raised as an objection. What if even though the tag name is the same, the elements themselves are different? This is a justified objection

to the example above. And a conclusion is, that it it is up to the developers to use the standard properly.

Assuming a common micro frontend landscape where every micro frontend is developed by independent, isolated teams, using heterogenic frameworks, the mentioned issue might occur. Two teams use Web Components and try to register different elements under the same tag name in the same landscape. This requires organizational interference on team level.

One way would be to define a name space for every team when creating Web Components e.g. Team 1 has the name space `team-1`. Making a Web Component registered by the micro frontend of Team 1, be named `team-1-tagname`. That again could lead to redundancies, because there is no way of assuring that the `team-1-tagname` and `team-2-tagname` Web Components are not the same.[43]

A better way might be to assign a common Web Components library like **UI5 Web Components**. That way the registered elements are provided by the library and are limited to set of unique registrable tags. If another micro frontend would use an already registered element of the library, a warning is thrown but no error occurs.

Most Web Component libraries also offer so called scoping options. This feature finds its use case, when versions of the used components differ from micro frontend to micro frontend. This option enables the developers to customize their Web Components and register them under different tags. With this feature it is made possible to register components according to their respective versions, by adding a version-specific suffix. That of course might lead to redundancies again. But it also reduces the dependency of the developer teams to always use the latest version of the component library or the first registered element in the landscape. [11] [27]

2.4 Webpack Module Federation

A rather new way of creating micro frontends is the **Webpack 5 Module Federation (WMF)**. With the release of version 5 in October 10 in 2020, this technology added certain features which improved its usage for developing micro frontends.[35]

Since the main focus of this document is to propose methods of avoiding redundancies in micro frontend landscapes, an introduction of this WMF feature is given here.

The basic concept of the new WMF involves so called `hosts` and `remotes`. These terms are comparable to the **Luigi Core** and **Luigi Nodes** principle, previously introduced in section 2.1. In case of WMF the `host` aka `shell` represents the **Core** or basis of

the micro frontend landscape and the `remotes` or `micro frontends` are the **Nodes** integrated or loaded into it. Still a direct comparison is not entirely possible since both frameworks look similar functionality-wise, but in their cores work differently. For instance, when using WMF one is tied to use the Webpack bundler, since the necessary configuration is done in the `webpack.config.js`. This restriction is applied to every micro frontend in the WMF micro frontend landscape, not only to the `host`.

WMF can be used in combination with most of the common UI Frameworks. But since the implementation for this thesis was done with Angular, the further examples and explanations will be Angular-based.

As explained WMF is a micro frontend framework, but for the purpose of solving the issue introduced in this transcript, it provides functionality too. This feature is enabled and configured, as well as the rest of the WMF, via the `webpack.config.js`. When configuring the components of the landscape, it is possible to define a section where shared dependencies are described. These dependencies can be defined in different ways. For instance, it is possible to define a strict version of the dependency, which would result in the framework loading this specific version. Or one can define a less restricted dependency which would mean, that if another `remote` loads the same dependency but in a different version, the framework would automatically apply the highest major version of the dependency to both micro frontends.

```
1      shared: share({
2        "@angular/core": {
3          singleton: true,
4          strictVersion: false,
5          requiredVersion: '12.2.0'
6        },
7        "@angular/common": {
8          singleton: true,
9          strictVersion: false,
10         requiredVersion: '12.2.0'
11       },
12       "@fundamental-ngx/core": {
13         singleton: true,
14         strictVersion: false,
15         requiredVersion: '0.33.0-rc.214'
16       },
17
18       ...sharedMappings.getDescriptors()
19     })
```

LISTING 2.3: Example of sharing dependencies configured in the `webpack.config.js`

The listing 2.3 is an example of how to share libraries in a restrictive way. To provide a less restricted configuration a simple array of the shared dependency names suffices. But to ensure a redundant free landscape, these restrictions are necessary. Each configuration property will be explained below.

- `singleton` - This property defines if the dependency should be able to be loaded more than once in different versions or not. If set to `true`, WMF will automatically pick the highest version of a major release of this dependency available and distribute it to the `remotes`.[32]

- `strictVersion` - This property defines if the dependency requires a specific version to work. If set to `true` WMF, will load the required version even if another dependency mapping with the same name is present. This can lead to conflicts with the `singleton` property, if configured poorly.

- `requiredVersion` - This property defines the required version of the dependency. When working with a package manager (e.g. NPM), this version has to be aligned with the locally installed version of the dependency. If the `strictVersion` property is set to `false`, this property defines the minimum version for the micro frontend.

It has to be mentioned that, WMF is able to distinguish between major releases. If a higher version of the same major release is available it will be loaded (e.g. `@angular/common@12.3.1`). For instance, if the next higher version is of a different major release e.g. 13.X.X, WMF would not consider to load it for the `remotes` which have the `requiredVersion` of release 12.X.X configured.

Now when it comes to sharing the dependencies of inside the micro frontend landscape, each `remote` has to bulk in. That means each micro frontend has to define its required dependencies in their respective versions. Additionally it has to be mentioned that the micro frontends themselves have to use dynamic imports when importing shared dependencies. Through the asynchronous behavior of the import, Webpack has time to pick the correct version of the dependency inside the landscape.[35] Analyzing this statement in combination with the information in the above list, it becomes obvious that multiple versions of the same framework can exist in a landscape. The below figure explains it in a graphical way.
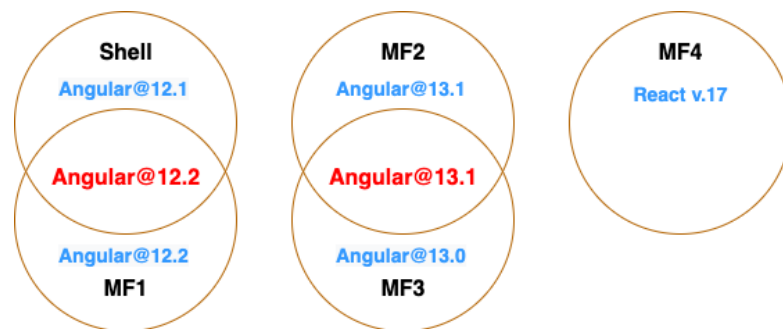
FIGURE 2.3: WMF way of handling multi-versions

As it can be seen in 2.3 WMF always picks the the highest major release, assuming the respective micro frontends have the above shown configuration 2.3 applied. If for instance, if **MF3** has its `strictVersion` property set to `true`, it would cause the loading of its libraries too.

When now looking at the introduced example, it is obvious that sharing several versions of the same framework over the whole landscape can not go without side effects. One of which is the increase in bundle sizes, since every `remote` bundles their local dependencies. WMF then picks the one to serve during the runtime of the landscape. This impact has a trade-off tough. Returning users can benefit from cached dependencies but in certain use cases this is not acceptable.[33]

3.   Content Delivery Networks

This chapter will give further information about the CDN technology and how it was implemented for the context of this thesis.

3.1   CDNs in general

CDNs these days are commonly used on the Internet. This is due to the fact, that CDNs have become a convenient way of providing the resources a website needs in split seconds, thus increasing its performance and user experience.[19] The following sections will provide a better insight of the CDN technology.

3.1.1   Architecture

The basic architecture of a CDN can be split into three different building blocks.

- **Point of Presence (PoPs)** - Strategically located data centers around the world. Their function is to reduce the round trip time of requests. PoPs usually consist of several caching servers.

- **Caching servers** - These servers are located in different PoPs and serve the function of caching resources from the origin server. That way website loading times and bandwidth allocations are reduced. This unit consists out of the actual hardware.

- **Hardware, like SSD/HDD and RAM** - Located in the cache servers, the purpose of this building block is to provide the necessary storage and computing capacity. Better hardware means faster computing time, which then again affects the overall performance of the designated caching server.

Besides the above three building blocks, another one is crucial for the architecture of a CDN. The **origin server** serves another purpose than a caching server. In a CDNs topology the origin server can be compared to the center or core. This is the server onto which the CDNs content is uploaded, synced with or distributed over the CDNs caching servers.[18]

Following these information, a basic CDN distribution can look as shown in 3.1.
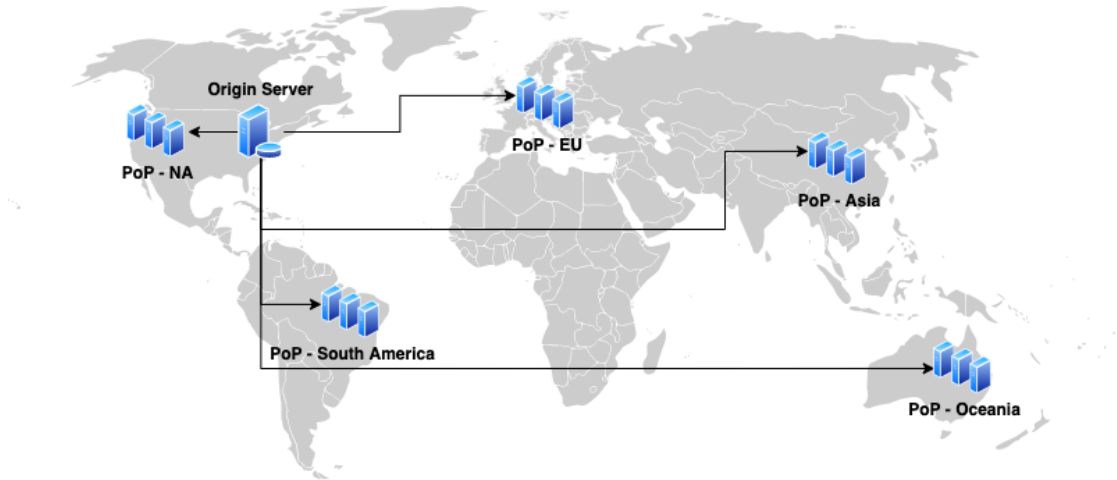
FIGURE 3.1: Basic distribution a CDN

It has to be mentioned that the geographical distribution in figure 3.1 is simplified, for the sake of readability. The cache servers of each PoP would be distributed themselves over different data centers on the continent. A client requesting resources from the CDN will be communicating with the nearest cache server to their designated location.[19]

3.1.2 Design

For a CDN to fulfill its purpose, four requirements have to be met by it. These are not only technical requirements like necessary hardware, but rather concerning the general design of the CDN. The four pillars of CDN design are mentioned below.

- **Performance** - First and foremost, the CDN has to provide a benefit when it comes to website performance. When the usage of a CDN increases the loading time of websites, the effect can affect the user experience and cause financial harm to the host.[19]

- **Scalability** - Since an cache server can serve multiple resources to different websites, it has to be able to handle traffic peaks. Without the aspect of scalability (either horizontal or vertical) these requirements can not be met and would affect the CDNs performance.

- **Reliability** - A website host has to rely on the CDN to deliver the websites resources. An outage on CDN side would cause the same effect to the relying websites, resulting in additional costs for the designated hosts. Therefore, CDN providers commit to 99.9% service level agreements (SLAs).

- **Responsiveness** - The aspect of responsiveness targets the issue of synchronization inside the CDN. A CDN has to be capable to react to changes and distribute

those accordingly to its PoPs over the globe. Otherwise inconsistencies could occur on the websites relying on the CDN. This is ensured through an automated pull mechanism, via which edge servers pull changes from the origin server.[18]

Additionally to those four requirements, the topology of a CDN has to be considered. There are two options which can be used and both would serve their purpose but bring some advantages as well as disadvantages.

- **The Scattered CDN** - This topology focuses on physical proximity. The PoP sizes are kept rather small but scattered around the world more frequent, thus providing as much proximity to their client as possible. This topology excels in providing the CDNs resources into low-connectivity regions, since it is not highly dependent on wire infrastructure. When it comes to latency, it does not suffer as much due to the short distance between client and server. The trade-off with this topology is, that it requires more maintenance costs since more PoPs are available to maintain. Also deploying new configurations can be connected to a lot of effort, depending on the amount of PoPs, scattered over the CDN. Additionally this amount can also affect the round trip time (RTT) , since every PoP inbetween the client and the server is a connection point.

- **The Consolidated CDN** - Other than the scattered CDN, this topology is designed to consolidate its resources at strategically located data centers. Since the PoPs are only located in those major data centers, the server available to the PoPs are highly advanced and provide a lot of hardware capacity. Additionally since the amount of major data centers around the world are rather limited, the amount of PoPs is diminished as well, compared to the previous topology. Following the quality over quantity principle, a PoP in this topology can handle more amounts of traffic compared to its counterpart and is also more resilient specifically when it comes to DDoS attacks. Also due to moderate amount of PoPs it is easier and faster for the operators of the CDN to deploy new configurations. Nonetheless, the trade-off for this topology is that even though it can handle a high amount of requests, its reach is rather limited to low-connectivity regions. This is due to the proximity difference between the servers and clients. Also deploying a new PoP into this topology is connected to more effort, since the PoPs are rather complex.

The design decision is of course dependent on the business case for the CDN, as both topologies mean to solve certain issues or challenges. [15]

3.1.3 Optimization

The CDN technology offers several ways of optimizing a websites performance and therefore, increasing the user experience. In the following section these optimizations will be showcased and explained.

3.1.3.1 Route optimization with Anycast

As previously mentioned, a CDN can be designed with different topologies which could affect its performance. But another factor which has to be considered in this equation is the routing itself. Basically it does not matter if a cache server proximately close located to a client, if the requested resource is not present there. In this case the client would have to be routed to different, further located cache server. Thus the Anycast routing is used in modern CDNs. This traffic routing algorithm is best explained in direct comparison with Unicast. Both serve the same purpose of routing requests to their designated destination. But they do it in different ways. Where with Unicast each node has a unique address, Anycast advertises multiple nodes with the same address. For instance, in an Unicast orchestrated network the server address `10.10.0.1` would be only present once. Anycast on the other hand would advertise this exact address over multiple different servers around the globe. Thus a request towards the address would reach its destination via the shortest path, given that the path will be identified and prioritized by devices that actually govern the flow of traffic. The shortest path itself is counted in hops. Hops represent the amount of time a request changes hands between hosts.[16]

3.1.3.2 TSL Performance

The route optimization with Anycast, does also improve the round trip time (RTT), when using the TSL/SSL protocol. Since this section does not focus on explaining the protocol, only a short description is given. SSL (Secure Socket Layer) or as it now should be called TSL (Transport Layer Security) is a protocol via which secure communications are ensured on the Internet. The communicating parties establish a connection via following steps:

1. A so called three-way handshake is done
2. The parties agree upon an encryption method
3. Mutual verification process is performed
4. Generating symmetric keys for encoding and decoding

These steps are necessary to ensure secure communication and are a welcome trade-off for the benefits they provide.

Even though it is an overhead and increases the RTT of a request, a CDN can provide improvement. Through the aspect of route optimization and the general proximity the overall request distance is decreased. Therefore, the RTT is shortened as well. The steps are still processed, they just do not have to travel that long. Additionally the SSL/TSL negotiation process is shorter too.[14]

3.1.3.3 Frontend optimization

The term frontend optimization refers to the process of making a website more browser-friendly and reduce loading times. There are multiple ways of to optimize a frontend. These will be explained under consideration of the role a CDN plays in them.

- **Reducing HTTP requests** - When loading a website the browser opens several HTTP connections. The amount of which are actually limited to the browser. If a website requires more connections than a browser can open at the time, the browser has to start queuing the rest. This again leads to longer loading times and affects the user experience. A CDN improves on that via pre-pooling connections and ensure they remain open throughout a session. Even though this does not reduce the actual number of requests, it does improve the response time for each one. Making it so that every request can be processed faster. Additionally HTTP/2, though it is in its early stage, introduces the method of multiplexing. This allows a single TCP connecting to transfer multiple different HTTP requests.

- **File compression** - Of course it is not always about the amount of requests per side loaded or the proximity of the client to the server. The actual content does affect the responsiveness too. Loading one single resource with a size of 1 GB takes a while even if the server is directly nearby. Reducing the size of this file or resource might increase the loading process. File compression like gzip is a method of doing exactly that. Most modern CDN providers offer automated file compression with gzip to reduce the actual content size delivered to the client.

- **Cache optimization** - Via caching static files are stored, either on the client device or in cache of the nearby cache server. Locally stored static files do not have to be loaded via the network and are available to the browser almost immediately for rendering. The only question remaining is, how long does a resource have to be cached? The answer to that is necessary, for optimizing the use of the clients cache. The caching time is usually defined in the cache header of the request. Modern

CDNs offer cache control options, which help in defining rules for exactly that header. CDNs also have started using machine learning techniques to follow and understand content usage patterns and automatically optimize caching policies.

- **Code minification** - Correlating with the file compression method, the process of code minification offers a way of reducing file sizes too. Where a developer writes code in a humanly readable way, with spaces and line breaks, a machine does not need this formatting. By removing comments, spaces and line breaks, the size of code file can be reduced by 30%. CDNs use methods like gzip, minify or these two even in combination to reduce the size of JavaScript, HTML or CSS files.

- **Image optimization** - Images can be immense in size and require a long time to load. The best way to display an image on a website would be to cache it first and then load it from the cache to reduce actual loading time through the network. Another option could be, reducing the actual size of the image and thus the loading time. Other than code files, images are already compressed when loaded, therefore compressing them further to reduce file size might cause a loss of image quality. This is called lossy compression. If this trade-off is not an option, caching would be more effective. CDNs offer exactly that solution, caching images and providing them from the nearest source available to the client. If this does not suffice, CDNs also offer a progressive rendering option for images. On initially loading the page, the CDN would provide a lossy compressed version of the image fast and then progressively replace it with higher-resolution variants of it. Alternatively a website host could use vector or raster images. These are resolution independent, smaller in size and highly responsive.

These methods in combination with the CDN technology provide a possible gain in user experience for a website host. [17]

3.2 Unpkg

As described in 2.2 the concept of CDN is fairly simple. A remote server is providing the necessary platform resources via an API, thus avoiding the necessity of bundling those resources. To evaluate the gain and impact of this technology on a micro frontend landscape, a prototype was developed using the public cloud based CDN Unpkg.com.

Figure 3.2 shows the Unpkg.com architecture, as it is described in its official documentation. It is an open source project built and maintained by Michael Jackson. It runs on the Cloudflare platform and the Fly.io infrastructure provides its servers with auto-scaling capabilities on 17 cities around the world.[40]
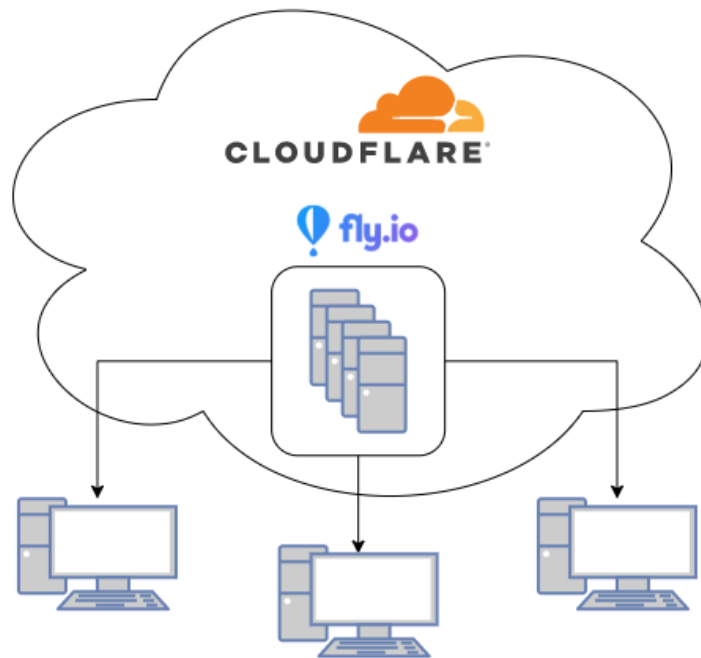
FIGURE 3.2: Basic architecture of the unpkg.com public CDN

An open API is available, through which resources can be requested. Via path and query parameters in the URL, necessary information like the dependency version can be provided.

The previously mentioned prototype was developed using Unpkg.com. The figure 3.3 visualizes the architecture of the prototype.
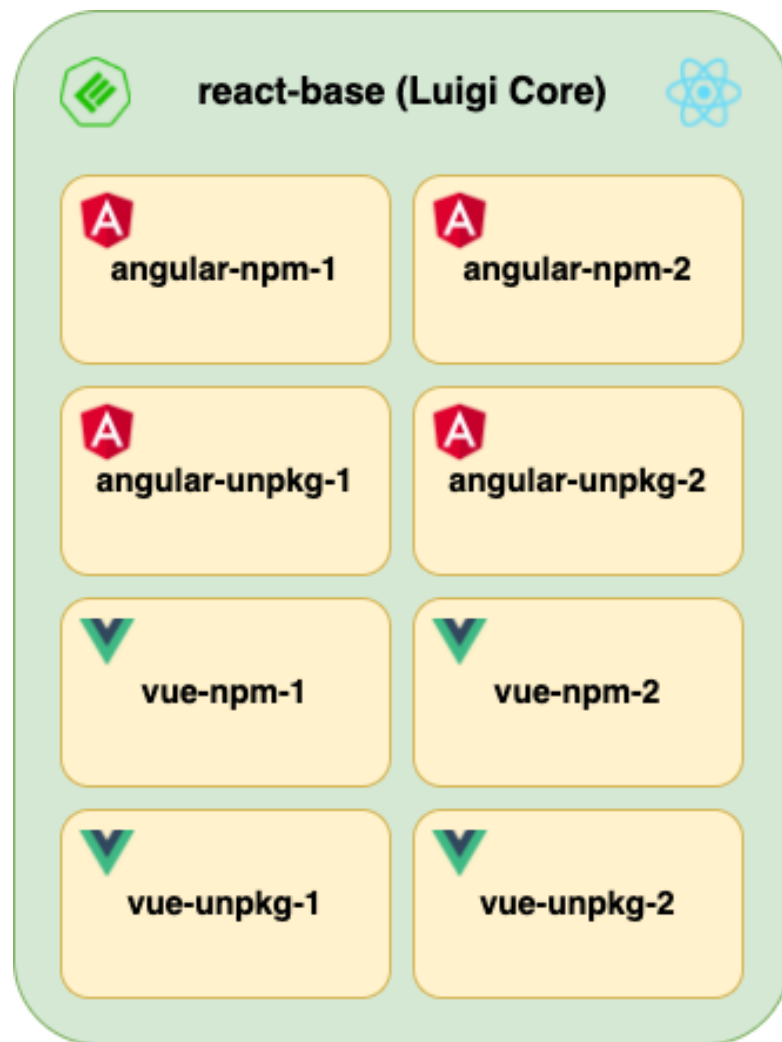
FIGURE 3.3: Architecture of the Luigi prototype using unpkg.com

Several Nodes were embedded in the Luigi landscape. Some of which are Angular apps, other are implemented with Vue. It has to be added, that apart from the difference regarding the UI-Frameworks, each micro frontend displays the same elements. This design decision was made, to ensure a base for later comparisons. Additionally, each micro frontend was implemented using a regular bundler and package manager. This way a direct comparison is possible between the two technologies.

The usage of the Unpkg-CDN itself is embedded in the code. Instead of loading the dependencies via the local `node_modules` directory, they are directly loaded using the CDNs API. The usage of which is displayed in the listing 3.1.

```
1  <script src="https://unpkg.com/@ui5/webcomponents@1.0.0-rc.15/
      dist/StandardListItem.js?module" type="module"></script>
```

LISTING 3.1: Import of a dependecy using the unpkg API

This script tag is placed in the central `index.html`. The loaded resource can be read from the URL. First part of the URL are obviously the protocol and the host. The first path parameters is the npm dependency name. In the case of 3.1 it is `@ui5`, followed by a subdirectory with its respective version annotated with an @. As it is described in the documentation, the CDN supports two query parameters.

- `?meta` - to request meta data about the loaded package, in a JSON format
- `?module` - to expand all import specifiers in JavaScript modules to unpkg URLs

If the import is handled via a script tag as it is done in 3.1, the `type="module"` attribute has to be added. This is necessary depending on the resource loaded. In the case of the given example, the resource is a JavaScript module type. To ensure it is parsed by the runtime as such, this attribute has to be added.[9]

3.3 Owning a CDN

Even though the method of a self-hosted CDN, was not implemented in the developed prototype, it has to be considered for the context of this thesis. Since the basic principle of a CDN was already explained, this section will focus on the financial aspect of the technology.

For the purpose of the analysis, the following scenario is assumed.

- A micro frontend landscape has to be developed.
- To reduce the runtime costs, the loading of redundant libraries should be avoided in the landscape.
- The libraries for the landscape were developed by the team and are only available for internal use.

If now the decision to use a CDN is made, only a self-owned one would be applicable.

In the context of the given scenario following parameters are also assumed or averaged, based on existing products for further cost-calculations. The micro frontends of a Luigi landscape which are requesting resources from a CDN have following metrics defined. Some of which are based on average values from the implemented prototype.

- How often the site is accessed per day - $50000 times$
- Avg. byte size per page load without caching on client side - $8983.5 KB$

- Avg. amount of GET requests to CDN per side load - 198 requests
- The site is only accessed in North America

Taking this metrics, following values are calculated for the monthly usage of the landscape.

Let A be the amount of requests to the CDN per month:
$$A = 50000 \times 30 \times 198 = 297000000$$

Let B be the amount of bytes loaded per month:
$$B = 50000 \times 30 \times 8983.5 = 13475250000KB = 13.47525TB$$

In the following text, it will be analyzed, how much it would cost a company or department to pay for a CDN, hosted by one of the common CDN providers.

There are multiple CDN solution providers on the market. Some of which are **Amazon CloudFront, Azure CDN, Google Cloud CDN**. [2] Since the pricing models of the mentioned providers differ, it´s hard to compare them directly. Applying the given scenario to the pricing calculator of the respective providers results in the following numbers.

- Amazon CloudFront - 1426.22$ per month, for North America
- Google Cloud CDN - 1168.99$ per month, for North America
- Azure CDN - 1123.90$ per month, for North America

After averaging those values following *Avg.price* is calculated.

$$Avg.price = \frac{(1426.22 + 1168.99 + 1123,90)}{3} = 1239.70\$$$

It has to be mentioned, that these values were calculated using the basic packages offered by the providers. There are further features which can be added to the solutions, that of course would increase the sum. Additionally other providers have different pricing models, some of which include pricing depending on the HTTP requests sent to the CDN. Others charge prices when resources are stored in several Points of Presences (PoPs) around the world. Also locations play a key role, different regions pay more for traffic then others and this value is provider specific. Since it is not the goal to pick a provider, but rather provide a general overview of costs for such a solution, those additional features were not considered for the given scenario.

4. Web Components

4.1 Web Components in general

As in 2 introduced, Web Components have the attribute of the Custom Element, via which the individuality of each registered tag is guaranteed. Nonetheless to understand the actual functionality of Web Components, it is necessary to explain the other aspects of it. The following section will introduce the other three standards of Web Components and their functions and showcase how exactly they serve a purpose for the context of the thesis.

4.1.1 Shadow DOM

The DOM (Document Object Model) represents the elements of a markup document in a tree-like structure, consisting of connected nodes. The commonly used markup language for websites is HTML. [6] The Shadow DOM is also a DOM, but is attached to the actual DOM of the document. Underneath it, elements can be defined the same way as they are in the regular DOM. The difference appears during the rendering of the document, when a page is loaded. The Shadow DOM elements are rendered separately from the DOM it is attached to.[30]

To understand the relationship between the two connected DOMs following terms have to be explained.

- **Shadow host** - The attachment of the Shadow DOM to the normal DOM happens via a node inside the normal DOM.

- **Shadow tree** - Since the Shadow DOM is a DOM in itself, it consists of nodes in a tree-like structure.

- **Shadow boundary** - The Shadow DOM capsules its Shadow tree an renders it separately from the actual DOM. This encapsulated area defines where the Shadow DOM begins and ends.

- **Shadow root** - Just like a regular DOM a Shadow DOM has a root from where it originates.

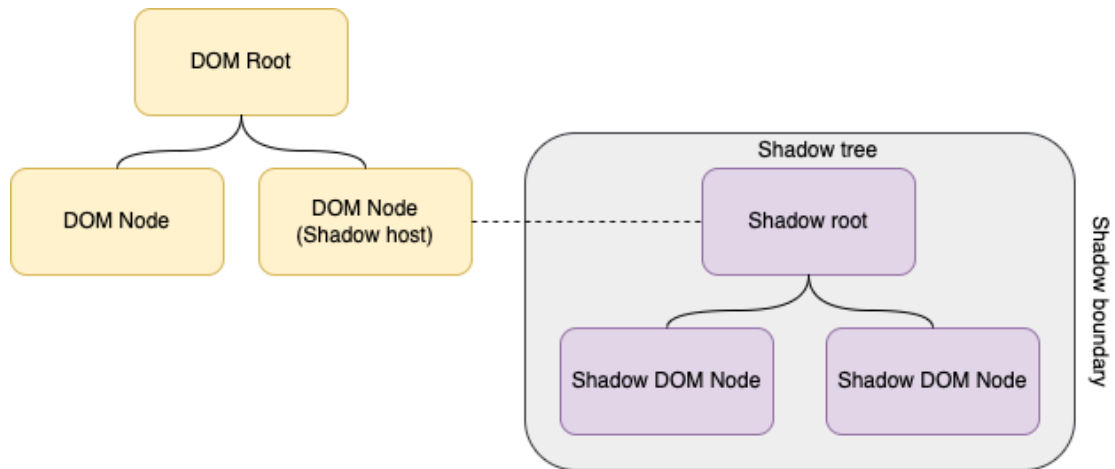Figure 4.1 visualizes the relations between the newly introduced terms.

FIGURE 4.1: Shadow DOM architecture

Through the isolation of the Shadow DOMs code, this standard offers a way to provide scoped HTML and CSS code to custom elements. As mentioned before, the nodes of the Shadow DOM are rendered separately. Therefore, styles, ids, names or even CSS classes and other configurations applied to a tag inside the Shadow DOM are not applied to the actual DOM.

Listing 4.1 provides be a simple example.

```
1  <!DOCTYPE html>
2    ...
3    <body>
4      <p>Plain</p>
5      <custom-element></custom-element>
6
7      <script>
8        class CustomElement extends HTMLElement {
9          constructor() {
10           super();
11           const shadow = this.attachShadow({mode: 'open'});
12           shadow.innerHTML = `<style> p {color: red;} </style>`;
13           const shadowParagraph = document.createElement('p');
14           shadowParagraph.textContent = 'Blue';
15           shadow.appendChild(shadowParagraph);
16         }
17       }
18       customElements.define('custom-element', CustomElement);
19       customElement = new CustomElement();
20       console.log(customElement.shadowRoot)
21     </script>
22   </body>
```

```
23  </html>
```

The listing shows the usage of the Shadow DOM in combination with registering a custom element (named `custom-element`) using the customElements API which was explained in chapter 2. The `script` tag of the snippet starts with creating a new HTML element called CustomElement. It is an extension to the HTMLElement class. This new element has the Shadow DOM attached to it. Therefore, it is the Shadow host in the DOM tree. Via the `innerHTML` attribute a Shadow DOM Node is created. In this case it is a styling for paragraphs,followed by the creation of a paragraph onto which the styling is applied. Additionally the text color of the paragraph is set to blue and lastly it is attached to the Shadow root. The last three lines of the script, are the definition and registration of the new `custom-element`, the creation of a new `customElement` object and the output of the attached Shadow DOM in the console.

Now when looking at the body of the above snippet, it becomes visible that beside the `custom-element` another paragraph is defined in the document. Usually the styles defined in the `script` below would apply to this paragraph, too. Through the encapsulation of the two separate DOMs this is not the case. The paragraph defined in the body of the document will be completely unaffected by the changes applied to the paragraphs in the Shadow DOM.[30]

Now another configuration which is shown in the snippet is the `this.attachShadow(mode: 'open')`. This mode property defines the visibility of the Shadow DOM to the browser. If set to `open`, the elements of the Shadow DOM can be inspected inside the sources. Thus, the `console.log(customElement.shadowRoot)` line would work if executed. If the mode would be set to `closed` the `console.log()` command in line 20 would return `null`.[30]

This behavior is not meant to be used as a measure of security, since it can be overwritten. The Shadow DOM encapsulates every part of its DOM elements, that means HTML, CSS and JavaScript. The `document` object, available during runtime, stays the same for the regular DOM as for the Shadow DOM though. Therefore, each configuration done in the Shadow DOM via scripts can be easily overwritten from any other script in the document, thus the mode can be changed even if initially set to `closed`.[44] For Web Components in particular it is not recommended to use this mode at all, since it would make them less flexible for end users.[4]

This feature can be used in combination with a component library like UI5 Web Components, to customize styles or appearances of individual components. Nonetheless, for the context of this thesis, it does not serve any purpose to actually avoid redundancies in

a micro frontend landscape, instead it might even increase them. The encapsulation of code elements between the single components increases the isolation and thus it cannot be guaranteed that e.g. the same CSS class is not rendered in multiple different Shadow DOMs. Even though the redundancies would occur on code level and not on dependency level, they are redundancies nonetheless.

4.1.2   HTML Template

This standard offers a way to define reusable markup code. As a part of the HTML standard itself, the `template` tag is used to define templates. These are not rendered unless used by another element. Similar to `template`, `slot` serves the same purpose, but in a different way. Templates are defined HTML code snippets which can be cloned and inserted in other document elements or even elements rendered in the Shadow DOM. Slots on the other hand serve as placeholder for either default markup texts or other DOM elements. Therefore, a template is a rather static piece of reusable HTML code, compared to slots. Slot themselves are identified by their name, the content is inserted when the slot is addressed by its name. Listings 4.2 and 4.3 provide examples how exactly these two standards are used. [7]

```html
<!-- Definition of the template -->
<template id="my-paragraph">
  <p>My paragraph</p>
</template>

<!-- Usage of the template in an Web Component -->
customElements.define('my-paragraph',
class extends HTMLElement {
  constructor() {
    super();
    let template = document.getElementById('my-paragraph');
    let templateContent = template.content;

    const shadowRoot = this.attachShadow({mode: 'open'})
    .appendChild(templateContent.cloneNode(true));
  }
});
```

LISTING 4.2: Definition and usage of the `template` standard [7]

It is important to note, that the defined template in the listing is not rendered unless somehow included in a DOM (either Shadow DOM or the regular DOM), via Java Script.

```html
<!-- Definition of the slot -->
```

```
2   <p>
3     <slot name="my-text">Default input</slot>
4   </p>
5
6   <!-- Usage of the slot in the markup document -->
7   <my-paragraph>
8     <ul slot="my-text">
9       <li>Some different input</li>
10      <li>In a list!</li>
11    </ul>
12  </my-paragraph>
```

LISTING 4.3: Definition and usage of the `slot` standard [7]

As it can be seen in 4.3, the `slot` definition has some default content defined. In the above case it´s a simple text. When the slot is used, this content is overwritten by the content in the element which is calling the `slot` by its name. In that case the content is replaced by some list items. Other than the templates, slots are always rendered if included in the markup via their respective names. The content which is rendered is depended on, if the default content is overwritten or not.

Even though slots are a HTML standard just as templates, their support for browsers is not always guaranteed. This is due to the fact, that compared to templates it is a rather new standard.

In combination these two standards offer a way to define flexible, reusable markup code for Web Components.[7]

When developing own Web Components for a micro frontend landscape, this aspect can be used to reduce repetitive HTML code elements by serving them a templates. But again this feature is not applicable when it comes to reducing redundant libraries in a micro frontend landscape. Redundant code snippets can be defined as reusable templates but not the actual libraries.

4.1.3   ES Modules

The last standard is not referred to by every source, but according to [42], it is a stable part of the Web Components standard, via which Java Script modules can be defined and reused by other documents. Thus the development of Web Components can be done in a modular way, making every component available to other documents, using the `type="module"` attribute.

```
1  <!-- Import of the JS Module -->
2  <script type="module" src="awesome-explosion.js"></script>
3    ...
4  <script type="module">
5    import 'awesome-explosion.js';
6    ...
7    import {awesomeExplosion} from '@awesome-things/
     awesome-explosion';
8  </script>
9
10 <!-- Usage of the newly imported module -->
11 <awesome-explosion>
12   ...
13 </awesome-explosion>
```

LISTING 4.4: Importing modular Java Script documents into another [42]

Listing 4.4 shows such an import. Assuming the `awesome-explosion.js` files contains the definition of an element called `awesome-explosion`, these lines enable the document to use this element.[42]

This feature is essential for the prototype developed, since via this aspect the components are made available to the landscape. The used Web Components are served as ES Modules to the landscape and are imported similarly as shown in 4.4. Therefore, this aspect enables a component to be imported in multiple micro frontends as a module.

4.2 Usage of Web Components in the Prototype

The Web Components standard was partially used in the development of the prototype. This technology was used to embed the developed components as part of a compound view, using the corresponding feature in Luigi. [22] [21] For development itself no UI application framework was used, which means all three compounds were developed using plain Java Script. The actual components used in the micro frontends of the compounds, were taken from a component library called UI5 Web Components. Each element in that library is in fact a Web Component.[37] Consisting of the picked elements and under the usage of the scoping feature of the component library, the compounds where created. This made it possible to scope the tags of the UI5 Web Components.[11]

```
1  import { LuigiElement, html }
2  from "@luigi-project/client/luigi-element.js";
3  import { setCustomElementsScopingSuffix }
4  from "@ui5/webcomponents-base/dist/CustomElementsScope.js";
```

```
 5  setCustomElementsScopingSuffix("placeholder");
 6  import "@ui5/webcomponents/dist/Dialog.js";
 7  import "@ui5/webcomponents/dist/Button.js";
 8  [...]
 9
10  export default class extends LuigiElement {
11    constructor() {
12      super();
13      [...]
14    render() {
15      return html`
16      <div>
17        <div class="header">
18          <h2>Products table - Version: placeholder</h2>
19        </div>
20
21        <ui5-table-placeholder id="ui5-table"
22        ui5-table sticky-column-header>
23          <!-- Columns -->
24          <ui5-table-column-placeholder
25            slot="columns"
26            style="width: 4rem">
27            <span
28            style="line-height: 1.4rem">
29              Product
30            </span>
31          </ui5-table-column-placeholder>
32
33          [...]
34
35        </ui5-table-placeholder>
36      </div>`;
37    }
38  }
```

LISTING 4.5: Scoping feature used in the prototype

Listing 4.5 shows the exact usage of the scoping feature. The imported *setCustomElementsScopingSuffix* function enables to define a custom suffix to all UI5 Web Component elements, if not configured otherwise. Also, as it can be seen in the same listing, the suffix is set to *placeholder* using the imported method *setCustomElementsScopingSuffix("placeholder")*. This is done for the sake of the experiment itself. It was intended to deploy six same or similar looking Web Components into the Luigi landscape to test how many redundancies occur. In addition it was meant to be tested how different

versions of the same elements could be registered. For example the *Bar* element in version *1.0.1* is registered under same tag as the same element in version *1.1.0*. This scenario has to be handled and the scoping feature is one way of doing so. By defining a version suffix for the *Bar* element tag, the browser can distinguish those elements. In case of the prototype to create this exact scenario, one global tag suffix *placeholder* was picked and later replaced. The replacement itself happened during the bundling of the project. The RollUp bundler provided the necessary functionality for that. Inside the `rollup.config.js` several configurations were defined, which are shown in listing 4.6.

```
1   imp\part{ort resolve from '@rollup/plugin-node-resolve';
2   import json from '@rollup/plugin-json';
3   import url from "@rollup/plugin-url";
4   import { terser } from "rollup-plugin-terser";
5   import replace from '@rollup/plugin-replace';
6   import { SameVersions } from './rollup_files/same_version';
7   import { DiffVersions } from './rollup_files/different_version';
8   import { MixedVersions } from './rollup_files/mixed_version';
9
10  let buildArray = [];
11
12  function aggregateConfigs() {
13    for(let buildConfig of SameVersions) {
14      buildArray.push(buildConfig);
15    }
16
17    for(let buildConfig of DiffVersions) {
18      buildArray.push(buildConfig);
19    }
20
21    for(let buildConfig of MixedVersions) {
22      buildArray.push(buildConfig);
23    }
24  }
25
26  aggregateConfigs();
27
28  export default buildArray;
```

LISTING 4.6: Content of the `rollup.config.js`

The configuration of the bundler was split into several Java Script files, to improve the readability of the configuration file itself. The content of such a file can be seen in 4.7.

```
1   imp\part{ort resolve from '@rollup/plugin-node-resolve';
2   import resolve from '@rollup/plugin-node-resolve';
```

```
3  import json from '@rollup/plugin-json';
4  import url from "@rollup/plugin-url";
5  import { terser } from "rollup-plugin-terser";
6  import replace from '@rollup/plugin-replace';
7
8  export const MixedVersions = [
9    // 1
10   {
11     input: 'src/tableView.js',
12     output: {
13       file: 'dist/tableViewMixedVersions1.js',
14       format: 'es',
15       compact: true
16     },
17     plugins: [
18     replace({
19       'placeholder': '0-9-0',
20     }),
21     terser(),
22     resolve(),
23     json(),
24     url({
25       limit: 0,
26       include: [
27       /.*assets\/.*\.json/,
28       ],
29       emitFiles: true,
30       fileName: "[name].[hash][extname]",
31       publicPath: "\" + new URL(\".\", import.meta.url) + \"", //
     relative configuration for assets (TBD with UI5 Web Components
     team)
32     })
33     ]
34   },
35   // 2
36   {
37     input: 'src/tableView.js',
38     output: {
39       file: 'dist/tableViewMixedVersions2.js',
40       format: 'es',
41       compact: true
42     },
43     plugins: [
44     replace({
```

```
45        'placeholder': '1-1-0',
46      }),
47      terser(),
48      resolve(),
49      json(),
50      url({
51        limit: 0,
52        include: [
53        /.*assets\/.*\.json/,
54        ],
55        emitFiles: true,
56        fileName: "[name].[hash][extname]",
57        publicPath: "\" + new URL(\".\", import.meta.url) + \"", //
     relative configuration for assets (TBD with UI5 Web Components
     team)
58      })
59      ]
60    },
61    [...]
62 ]
```

LISTING 4.7: Actual configuration for the `rollup.config.js`

The same file is used for bundling and each time it is bundled differently. The `replace` method of to configuration object replaces a string inside the input file. In this case the string *placeholder* is first replaced with *0-9-0* and in the second bundle configuration with *1-1-0*. Also the files generated in the process are called differently, the first one is called `tableViewMixedVersions1.js` and the second `tableViewMixedVersions2.js`. Even though the elements inside those files are actually the same, they are handled as differently, since their tags differ in the browser. Elements register for the first view are called for example `<ui5-bar-0-9-0>` and for the second then `<ui5-bar-1-1-0>`.

That means out of the same file, several differently bundled and named files were generated. Each one of the bundled files register the same elements under different tags which are therefore treated as new elements.

5. Webpack Module Federation

Since an introduction of this technology was given in chapter 2, only a short recap of WMF is given here. Main focus of this chapter will be to showcase how the technology was implemented in the prototype. As mentioned before the implementation of the WMF was mainly done in combination with Angular. Therefore, all the examples and explanations below will have their focus on exactly that framework. It has to be mentioned though, that WMF is not restricted to Angular alone. The only restriction applied is concerning the bundler, which has to be Webpack.[35]

5.1 Short recap

WMF is a new way to create micro frontends using the Webpack bundler in version 5. The way it does that, is via modularizing self-compiled code parts and publishing them for integration by other modules. This published modules can be micro frontends themselves and are called remotes whereas the integrating modules are called hosts. Hosts refer to remotes under a configured name. This name is not actually known to the host during the compile time, but is first resolved at runtime. The self compiled remote in this case can be anything, a micro frontend or some sort of utility script. This way the Module Federation provides a way to avoid external or manual script loading and instead gives opportunities to automatically lazy load necessary code blocks during runtime.[35]

Integrating several remotes using the same dependencies leads to the issue of redundancies, which WMF is able to resolve. Via configuration of shared dependencies WMF provides a way to reduce redundant libraries in its landscape. A more detailed explanation was given in 2.

5.2 Implementation in the prototype

As previously mentioned the main functionality of WMF used in the prototype, is the shared dependencies feature. It is explained how it was implemented and an overview of the landscape is given.

For the WMF landscape of the prototype the Angular framework was used. The component library for the user interface was `@Fundamental-NGX/Core`. To showcase the handling of shared libraries and multi-versions in a micro frontend landscape with WMF,

several WMF projects where created and accordingly configured. For the rest of this chapter the WMF specific terminology will be used.

- **Same versions** - This is the host containing remotes of the same Angular version. The dependency itself is shared. This application was implemented to be used as comparison to the multi-version hosts.

- **Mixed versions** - This is the host containing partially the same and different Angular versions. Each version of the library is shared inside the landscape between. Purpose of this implementation was to showcase how many redundant versions of the same library are maintained by WMF in a multi-version landscape and how it affects the performance.

- **Different versions** - This host contains only different Angular versions. Therefore, it is an extreme version of the *Mixed versions* implementation. Every remote published requires a different version of Angular. Goal with this implementation was to showcase the performance losses in comparison to the other two landscapes. It rather serves as a negative example of how WMF should not be used.

To guarantee a basis for comparison, the user interface of each remote is the same. The figure 5.1 shows the elements displayed in each remote.

It contains a table with several table items in it. The angular version of each element is displayed above in the header.

5.2.1 The Webpack bundler

Prior to introducing the implemented landscape in detail, it is necessary to talk about Webpack itself first. As mentioned before, Webpack is a mandatory feature for using the Module Federation. Popular UI frameworks like React, VueJS or Angular use Webpack under the hood anyway, so it isn't as much of a restriction as it seems.[1][29][5] The documentation of the named frameworks imply that Webpack is used by default, but can be customized if necessary by the developer.

For Angular in particular, it is necessary to install two dependencies via the Angular CLI, to enable the features of the Module Federation. The command used for this is shown in 5.1.

```
1  ng add @angular-architects/module-federation --project name --port
      port
```

LISTING 5.1: Angular CLI console command to enable Module Federation in an Angular project

Products table - Same Versions: 12.2.0

Product	Price	Stock	Description
◎ Razor Gaming Mouse	999 EUR	12	Best mouse to flame people on League of Legends.
◎ Logitech Mouse	45 EUR	80	LIGHTSPEED Wireless Gaming Mouse with HERO Sensor
◎ Logitech Keyboard	50 EUR	22	A physical keyboard that uses an individual spring and switch for each key. Today, only premium keyboards are built with key switches; however, they were also used in the past, such as in the Model M keyboard from IBM, which used buckling spring switches
◎ HP Optical Mouse	35 EUR	4	Utilizing the latest optical sensing technology, the HP USB Optical Scroll Mouse records precise motion.
◎ Magic Mouse	40 EUR	20	The Magic Mouse 2 (Apple Magic Mouse 2), is a computer mouse developed and released by Apple Inc. It features a multi-touch acrylic surface for scrolling. ... The mouse features a lithium-ion rechargeable battery and Lightning connector for charging and pairing.
🖥 MacBook Pro	1299 EUR	11	It features a touch-sensitive OLED display strip located in place of the function keys, a Touch ID sensor integrated with the power button, a butterfly mechanism keyboard similar to the MacBook, and four USB-C ports that also serve as Thunderbolt 3 ports.
ƒx Brother Printer	235 EUR	24	Our affordable, quality machines provide you with the optimal way to take care of all your printing needs. Shop for the right printer, all-in-one, or fax machine for your home or home office today.
			34" U3419W Monitor, Display with stand

FIGURE 5.1: Example of an implemented remote in the prototype WMF landscape

These commands enable the Module Federation for an Angular project. Since the CLI protects the Webpack configuration from access, a custom builder is required. The `@angular-architects/module-federation` package provides exactly that. After installing this dependency in an Angular project, a `webpack.config.js` will appear on root level of the corresponding project.[36] This dependency has to be added in each remote or host of the WMF landscape.

After enabling the Module Federation inside a project, the necessary configuration can be applied to the `webpack.config.js` file. The remotes publish their modules and hosts consume them. Thus a developer can distinguish what module has which role. The examples of such configurations for hosts and remotes are shown in section 5.2.2, accompanied by the usage if the shared dependencies feature.

Since the implementation of the remotes is always similar, it won't be mentioned, instead the configurations of the corresponding `webpack.config.js` files will be explained in addition to the project structures.

5.2.2 Implementation of the prototype versions

The three implemented versions for the the WMF landscape are fairly similar to each other, the only difference can be found are the dependencies and their configured sharing. Listing 5.2 shows the configuration for the *sameVersion* environment.

```
1  [...]
2  module.exports = {
3    [...]
4    plugins: [
5    new ModuleFederationPlugin({
6      name: "shell",
7      filename: "remoteEntry.js",
8      shared: share({
9        "@angular/core": {
10          singleton: true,
11          strictVersion: false,
12          requiredVersion: '= 12.2.0'
13        },
14        "@angular/common": {
15          singleton: true,
16          strictVersion: false,
17          requiredVersion: '= 12.2.0'
18        },
19        ...sharedMappings.getDescriptors()
20      })
21
22    }),
23    sharedMappings.getPlugin()
24    ]};
```

LISTING 5.2: Content of `webpack.config.js` of the shell of the same versions WMF project

As it can be seen the shared dependencies are defined in between lines 22 and 24. Line 18 and 19 define the name of the application in the landscape and the name of the file after bundling. For the above case it has to be mentioned that the remotes are separate applications in their own runtime. Therefore, they have to be imported via the network, thus no `remote` property is configured in this file. To add the dynamically

loaded remotes, a service had to be developed which imports the remotes at runtime. This service is shown in listing 5.3.[31]

```
[...]
@Injectable({ providedIn: 'root' })
export class LookupService {
  lookup(): Promise<PluginOptions[]> {
    return Promise.resolve([
    {
      remoteEntry: 'https://angular-wmf-same-mfe1.surge.sh/
    remoteEntry.js',
      remoteName: 'mfe1',
      exposedModule: './Mfe1',

      displayName: 'Mfe1',
      componentName: 'Mfe1Component'
    },
    [...]
    ] as PluginOptions[]);
  }
}
```

LISTING 5.3: Content of `lookup.service.ts` for remote module loading in shell applications

This service serves the information of the remotely loaded modules to plugins for actual rendering. The rendering itself is done in a proxy plugin component. This component defines a plain template as some sort of placeholder for the remotes. [31]

```
1  [...]
2  @Component({
3    selector: 'plugin-proxy',
4    template: `
5    <ng-container #placeHolder></ng-container>
6    `
7  })
8  export class PluginProxyComponent implements OnChanges {
9    @ViewChild('placeHolder', { read: ViewContainerRef, static: true
        })
10   viewContainer: ViewContainerRef;
11
12   constructor(
13     private injector: Injector,
14     private cfr: ComponentFactoryResolver) { }
15
16   @Input() options: PluginOptions;
17
18   async ngOnChanges() {
19     this.viewContainer.clear();
20
21     const Component = await loadRemoteModule(this.options)
22     .then(m => m[this.options.componentName]);
23
24     const factory = this.cfr.resolveComponentFactory(Component);
25     const compRef = this.viewContainer.createComponent(factory,
      null, this.injector);
26   }
27 }
```

LISTING 5.4: Content of `plugin-proxy.component.ts` for remote module loading in shell applications

Line 8 of listing 5.4 defines the `ng-container` with the identifier called `placeholder`. This identifier is used in the code below to select and actually fill the container with a remote module. The functionality is placed in one of Angulars Lifecycle hook methods `ngOnChanges`, which is triggered when changes to input properties occur.[20] In there, the container is first cleared, then a remote loading option is selected from the array of the `lookup.service.ts`. The following line creates an Angular component out of the loaded remote and inserts it into the placeholder container, using the imported dependencies. The type of the plugin options was defined in an interface, exporting a type definition. The code for is displayed in listing 5.5.

```
1  import { LoadRemoteModuleOptions } from '@angular-architects/
      module-federation';
2  export type PluginOptions = LoadRemoteModuleOptions & {
3    displayName: string;
4    componentName: string;
5  };
```

LISTING 5.5: Content of `plugin.ts` for remote module loading in shell applications

The previously imported `@angular-architects/module-federation` dependency offers an existing type interface for that use case. This is extended by two more properties in line 4 and 5 of listing 5.5. By configuring the above service and component, it is made possible to load federated modules via the network into the host application. The configuration for a federated module can be seen in listing 5.6.

```
1  [...]
2  module.exports = {
3    [...]
4    plugins: [
5    new ModuleFederationPlugin({
6      name: "mfe1",
7      filename: "remoteEntry.js",
8      exposes: {
9        './Mfe1': './src/app/mfe1.component.ts'
10     },
11     shared: share({
12       "@angular/core": {
13         singleton: true,
14         strictVersion: false,
15         requiredVersion: '<= 12.2.0'
16       },
17       "@angular/common": {
18         singleton: true,
19         strictVersion: false,
20         requiredVersion: '<= 12.2.0'
21       },
22       "@fundamental-ngx/core": {
23         singleton: true,
24         strictVersion: false,
25         requiredVersion: '0.33.0-rc.214'
26       },
27       ...sharedMappings.getDescriptors()
28     })
29   }),
```

```
30    sharedMappings.getPlugin()
31    ]};
```

LISTING 5.6: Content of `webpack.config.js` of the mfe1 remote app of the same versions WMF project

As mentioned in chapter 2, to share dependencies every participant has to bulk in. Therefore, similarities can be found in the sharing configurations of the remotes and hosts. Between line 7 and 11, the actual federation of the module is configured. The reference to the module is later bundled in a file with the name defined in the `filename` property. In this case it is the `remoteEntry.js`. This is the file, which is automatically generated when the remote is compiled and serves as the entry point for the application when it is loaded into the host. Therefore this is the file accessed via the server url in the `lookup.service.ts` 5.3. As soon as the script is loaded via the service, the exposed module paths and names become known to the host and can be used to load the module. Thus the `expose` property contains a Java Script object, which maps the path to the actual component. In this case to `./Mfe1`.[35]

The implementations of the other versions are similar and only differ in the the shared dependencies configured. The effects of a multi-version environment and how it is handled by the Module Federation were explained in chapter 2.

6. Presentation of the results

In this chapter it will be explained how empirical traffic data was gathered using the prototype and which metrics were defined for comparison. Afterwards the landscapes will be compared, using the defined metrics as key performance indicators. The actual evaluation of the results will be concluded in chapter 7.

6.1 Data collection and metric definition

This section will contain information about how the data was collected and which metrics were defined based on the gathered data.

6.1.1 Data collection

6.1.1.1 Testing hardware and environment

It has to be mentioned that certain aspects of the environment were not maintained by the tester and are therefore out of reach for influence or configuration. Nonetheless they are mentioned here for the sake of reproducibility. The testing hardware and environment available were the following:

1. **Laptop:**

   - MacBook Pro (15-inch, 2017)
   - **Processor:** 2.9 GHz Quad-Core Intel Core i7
   - **Memory:** 16 GB 2133 MHz LPDDR3
   - **Graphics:** Radeon Pro 560 4 GB and Intel HD Graphics 630 1536 MB
   - **macOS Monetary - Version:** 12.2.1 (21D62)

2. **Browser, Network and Lighthouse:**

   - **Browser:** Google Chrome
   - **Version:** 99.0.4844.51 (Official Build) (x86_64)
   - **Chrome DevTools version:** Chrome 99
   - **Lighthouse version:** 100.0.0.0
   - **JavaScript version:** V8 9.9.115.8
   - **Network-Bandwidth:** 180-200 MB/s

3. **Runtime environment for prototypes:** surge.sh [1]

4. **CDN:** Unpkg.com [2]

Additionally to the specified hardware specs, version 1.18.1 of the `@luigi-project` was used. This dependency enables the Luigi framework and is therefore used in every implemented version of the prototype.

6.1.1.2 Testing process

Since multiple landscapes were implemented a unified process was required to collect comparable data. The BPMN diagram 6.1 contains the process steps in detail.
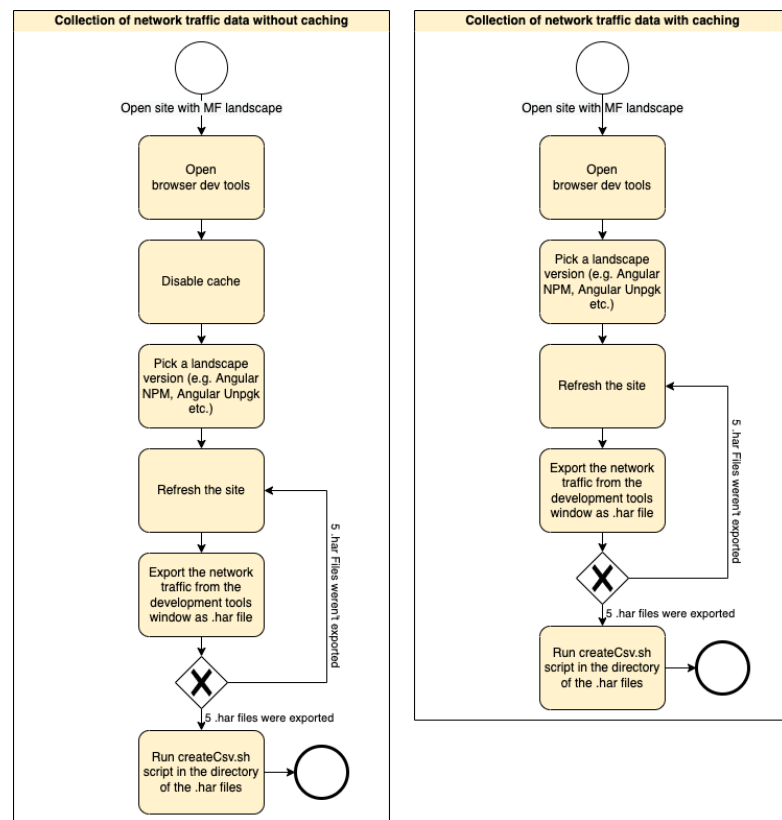
FIGURE 6.1: Data collection process for .har files

As it can be seen in 6.1 there are actually two processes executed to collect data. The first one was with caching disabled, the second had it enabled. The intention behind it was to gather an average initial loading time for the picked side, therefore the resources must not be cached. The second process was used to gather information about caching behavior in general. Since each landscape contains basically the same views, the browser

---

[1]surge.sh is a platform for static web publishing for frontend developers via the command line.
[2]This part of the environment was explained in chapter 3

should be able to pull already loaded resources from the cache instead via the network. To showcase this behavior and prove the performance gain through caching, this data was gathered too. Each process was executed 5 times, therefore 10 HTTP Archive (`.har`) files were generated per landscape. These `.har` files were then transformed into `.csv` files using a self-developed script.

Additionally to the `.har` files a website performance report was collected, using the Lighthouse tool embedded in the Google Chrome browser. Such a report provides information about how much of the imported bytes are actually used by a web application, distinguished by resource URL.

After the collection process, another self-developed script was used to aggregate all the gather data into one single file. This was later used to define and calculate the metrics, which are described next.

6.1.2 Defined metrics

The central file, containing all the data, was split into separate sheets in which the metrics for each micro frontend were calculated. The parameters for calculation were:

- **connection** - This is an ID with which each opened TCP connection is tagged by the browser. This value can be an indicator for the parallelity of the requests. Since HTTP/2 several requests can be handled by a server via the same TCP connection, reducing the time costs for TCP handshakes. It has to be considered that the application requesting the sources can not implement this HTTP/2 feature. It has to be enabled on server side and by the browser, which is already the case for most modern browsers.[13] As of writing this transcript, Surge the web server platform, onto which the application landscapes are deployed, does not support HTTP/2.

- **loadedFromCache** - This is a string value with three possible characteristics, *not loaded from cache*, *disk* or *memory*, whereas *disk* and *memory* are semantically the same. This value explains from where the resource was loaded. Either via the network from a remote server or from the local cache. It helps indicating the caching behavior of bundled resources in comparison with resources loaded from a unified URL (e.g. a CDN).

- **pageRef** - This is a page reference provided by the browser with possibly an infinite amount of characteristics. Each request is tagged with this value, so it can be distinguished, which request was sent by which web application. Since the data is separated by sheets inside the file and is always website specific, no further information can be pulled from this value.

- **startedDateTime** - This is a time stamp given to each request, marking its starting time in a DateTime format. This value can serve as an indicator for parallelity, since multiple TCP connections can be opened at the same time.

- **requestMethod** - This is a string value with a limited amount of characteristics. The possible values are the different HTTP methods. It indicates which type of request was executed for the loaded resource. In almost all cases it is **GET**.

- **requestUrl** - This is a string value, containing the request URL used by the browser to load the resource. This value is mainly categorizing the calculation results, since neither *connection* nor *pageRef* are valid options to distinguish the loaded resources of a website.

- **requestHeaderSize** - This is a number value, indicating the byte size of the request header sent by the web application to the server.

- **responseStatus** - This is a number value with a limited amount of characteristics, containing the HTTP response code from the server. The range of the possible values is the same as the possible response codes of the HTTP.

- **responseContentSize** - This number value is the byte size of the loaded resource.

- **timeInMs** - This value represents the round trip time of the request in milliseconds.

Based on the given values from the `.har` files, following key performance indicators were calculated.

- **Avg. Content Size** - Numerical value calculated by averaging the **responseContentSize** values of all the websites resources

- **Avg. Time in MS Loaded values** - Numerical value, calculated by averaging the **timeInMs** values of all the websites resources

- **Occurrences Connection Duplicates** - Numerical value to represent parallelity by counting the amount of duplicate/multiple occurring **connection** values

- **Connection IDs** - List of all **connection** characteristics for the website

- **Connection IDs occurrence** - Amount of how often the connection occurred during the loading of the website

- **Parallel start time** - Numerical value, calculated by counting reoccurring **startedDateTime** values

- **Avg. Response content size per Loading Type** - Numerical value, calculated by averaging the **responseContentSize**, differentiated by the characteristics of the **loadedFromCache** property

- **URLs loaded** - List of all unique **requestUrl** values

- **Avg. Response Time per URL loaded in MS** - Numerical value, representing the average loading time of each loaded resource URL of the website

- **Avg. Response Size per URL loaded** - Numerical value, representing the average loading byte size of each resource URL

- **Avg. Response Size per URL loaded via network** - Numerical value, calculated by averaging the **responseContentSize** differentiated by the **requestUrl** for all resources with the **loadedFromCache** value of *not loaded from cache*. For the test scenarios with cache disabled, this value equals **Avg. Response Size per URL loaded**

- **Avg. Response Size per URL loaded from Disk** - Numerical value, calculated by averaging the **responseContentSize** differentiated by the **requestUrl**, for all resources with the **loadedFromCache** value of *disk*

- **Avg. Response Size per URL loaded from Memory** - Numerical value, calculated by averaging the **responseContentSize** differentiated by the **requestUrl**, for all resources with the **loadedFromCache** value of *memory*

The main focus of this thesis is to reduce redundant libraries in micro frontend landscapes, by using different types of technologies. Therefore, the calculated metrics were used to partially determine the gain a technology brings. Nonetheless, it has to be considered, that certain indicators are hard to measure such as the complexity of the implemented technology. A metric like this, must not be ignored for the context of this transcript, since the gain or rather the value a technology brings, highly depends on it. Thus a subjective retrospective will be provided in chapter 7, describing the experience of the author with the used technologies.

In the following sections the results of the introduced metrics for each landscape are showcased. This display is distinguished by the fact, if the collected data was with caching enabled or not. Additionally to the metrics, the individual graphs for the landscapes are referred to in the appendixes A, B and C.

6.2 CDN landscapes

6.2.1 Implementation with NPM

The table 6.1 contains the results for the implemented landscapes with a regular bundler and a package manager, namely NPM. Certain metrics are not added to the table for readability, but a graphical representation can be found in A.

TABLE 6.1: Table of numerical KPI results for the NPM landscapes with caching disabled

Metric	Angular NPM	Vue NPM
URLs loaded count	14	13
Avg. Response content size per Loading Type	*not loaded from cache* - 129529.13, *memory* - 0, *disk* - 0	*not loaded from cache* - 156131.63, *memory* - 0, *disk* - 0
Parallel start time	33	45
Connection IDs occurence	*none established* - 10, *207393* - 10	*241394* - 10
Connection IDs count	62	71
Connection Duplicates	20	10
Loaded values & occurrences	*not loaded from cache* - 80, *memory* - 0, *disk* - 0	*not loaded from cache* - 80, *memory* - 0, *disk* - 0
Avg. Time in MS	162.22	5323.54
Avg. Content Size	129529.13	156131.63

The results shown in table 6.1 were collected without caching enabled. This explains for instance, why no resources were loaded from cache and therefore the values for the *memory* or *disk* characteristics are missing. The main takeaway from these results, is an average initial loading time for the landscapes which range from **129529.125** to **156131.625** milliseconds, depending on the framework. It can also be seen, that even though less resources are loaded via URLs by the VueJS landscape, it has a longer loading time compared to the Angular one.

TABLE 6.2: Table of numerical KPI results for the NPM landscapes with caching enabled

Metric	Angular NPM	Vue NPM
URLs loaded count	14	13
Avg. Response content size per Loading Type	*not loaded from cache* - 166856, *memory* - 17548.5, *disk* - 17548.5	*not loaded from cache* - 145646.67, *memory* - 0, *disk* - 187586.5

Table 6.2: continued from previous page

Metric	Angular NPM	Vue NPM
Parallel start time	32	39
Connection IDs occurence	*none established* - 20	*none established* - 20
Connection IDs count	61	61
Connection Duplicates	20	20
Loaded values & occurrences	*not loaded from cache* - 60, *memory* - 6, *disk* - 14	*not loaded from cache* - 60, *memory* - 0, *disk* - 20
Avg. Time in MS	150.90	1890.6
Avg. Content Size	129529.13	156131.63

The results of table 6.2 show the effect caching has on the performance of a site. The average loading time of the site decreases, as do the opened TCP connections. In table **??**, the Angular landscape is loaded faster compared to Vue JS.

Both environments were implemented using a regular bundler and therefore it can not be ensured that redundant libraries were not loaded.

6.2.2 Implementation with Unpkg

Table 6.3 contains the results for the implemented landscapes with a use of a public CDN, namely Unpkg. Certain metrics are not added to the table for readability, but a graphical representation can be found in A.

TABLE 6.3: Table of numerical KPI results for the landscapes using the Unpkg CDN with caching disabled

Metric	Angular Unpkg	Vue Unpkg
URLs loaded count	171	165
Avg. Response content size per Loading Type	*not loaded from cache* - 9768.62, *memory* - 0, *disk* - 0	*not loaded from cache* - 8205.19, *memory* - 0, *disk* - 0
Parallel start time	871	989
Connection IDs occurence	*223736* - 20, *223722* - 1574, *none established* - 10	*241394* - 20, *249654* - 1560
Connection IDs count	53	42
Connection Duplicates	1604	1580
Loaded values & occurrences	*not loaded from cache* - 1654, *memory* - 0, *disk* - 0	*not loaded from cache* - 1620, *memory* - 0, *disk* - 0

Table 6.3: continued from previous page

Metric	Angular Unpkg	Vue Unpkg
Avg. Time in MS	97.23	194.01
Avg. Content Size	9768.62	8205.19

The data shown in table 6.3 are again collected without caching enabled. An approximate estimation for the initial loading times can be drawn from this. It is made visible that the VueJS environment takes longer to load for less content. A similar behavior was present in the NPM implementation. Nonetheless it is an improvement in performance compared to the NPM counterparts for both landscapes. The trade-off of this technology is that single component resources, like a button or a table, were imported via script tags one by one, thus the increase in loaded URLs.

TABLE 6.4: Table of numerical KPI results for the landscapes using the Unpkg CDN with caching enabled

Metric	Angular Unpkg	Vue Unpkg
URLs loaded count	171	165
Avg. Response content size per Loading Type	*not loaded from cache* - 63972.08, *memory* - 208838, *disk* - 7166.1	*not loaded from cache* - 28143.2, *memory* - 0, *disk* - 7570.22
Parallel start time	1111	1069
Connection IDs occurence	*none established* - 1590, *223722* - 15	*none established* - 1570, *249654* - 10
Connection IDs count	52	42
Connection Duplicates	1605	1580
Loaded values & occurrences	*not loaded from cache* - 65, *memory* - 3, *disk* - 1587	*not loaded from cache* - 50, *memory* - 0, *disk* - 1570
Avg. Time in MS	58.73	87.1
Avg. Content Size	9762.72	8205.19

A similar development can be observed in table 6.4 as previously. Primarily with caching enabled, the average loading times decrease, secondly the amount of loaded resources from cache increases significantly and lastly the Angular environments show shorter average loading times despite more resources were requested by them.

6.3 Web Components/Compound and WMF landscapes

As the described in chapter 4, these environments exist in different versions. They were designed that way to showcase how the used technologies handle multi-version landscapes and how they affect the performance. Table 6.5 shows the above mentioned KPIs of those environments.

TABLE 6.5: Table of numerical KPI results for the landscapes implemented with Web Components as compound views and WMF with caching disabled

Metric	WC/Compound	WMF
URLs loaded count	27	118
Avg. Response content size per Loading Type	*not loaded from cache* - 213266.74, *memory* - 0, *disk* - 0	*not loaded from cache* - 507168.76, *memory* - 0, *disk* - 0
Parallel start time	29	109
Connection IDs occurence	*78725* - 15, *78890* - 15	*728247* - 3, *none established* - 14
Connection IDs count	546	42
Connection Duplicates	30	20
Loaded values & occurrences	*not loaded from cache* - 135, *memory* - 30, *disk* - 0	*not loaded from cache* - 563, *memory* - 0, *disk* - 0
Avg. Time in MS	2454.95	552.7
Avg. Content Size	213266.74	507168.76

As the data in table 6.5 shows, the initial loading time for the Compound/Web Components environments is significantly lower compared to the WMF counterpart. Also the amount of loaded resources differs. This is due to the fact, that WMF has to be used in combination with the Webpack 5 bundler. Therefore certain resources might be loaded several times, if not configured as shared dependencies in the Module Federation. Table 6.6 showcases the same landscapes, but with caching enabled.

TABLE 6.6: Table of numerical KPI results for the landscapes implemented with Web Components as compound views and WMF with caching enabled

Metric	WC/Compound	WMF
URLs loaded count	27	109
Avg. Response content size per Loading Type	*not loaded from cache* - 215385.36, *memory* - 167973, *disk* - 588082	*not loaded from cache* - 525243.08, *memory* - 0, *disk* - 1959
Parallel start time	35	105

Table 6.6: continued from previous page

Metric	WC/Compound	WMF
Connection IDs occurence	*none established* - 4, *78890* - 15, *78725* - 11	*none established* - 15
Connection IDs count	104	537
Connection Duplicates	30	15
Loaded values & occurrences	*not loaded from cache* - 107, *memory* - 20, *disk* - 4	*not loaded from cache* - 536, *memory* - 0, *disk* - 15
Avg. Time in MS	897.62	356.53
Avg. Content Size	219526.89	510997.6

The results with the caching enabled are not so different compared to the disabled caching ones. For instances, in both cases the average loading time for the compound landscape has longer loading times for fewer resources. One aspect differs though, compared to the compound landscape the WMF one has little cached resources. Also it becomes visible that the average content size loaded is smaller. Therefore, the Lighthouse tool was used to gather further information and data about those applications. The next section will showcase and explain the analysis done.

6.4 Lighthouse analysis

It has to be mentioned in advance that the following data is considered to be representative. Therefore, it is assumed that similar landscapes would follow a similar result in general. Nonetheless it is possible that for single cases the reports differ, since the within lying calculations are highly application dependent. For the sake of completeness the reports are shown below.

The given data was gathered based on the previously introduced landscapes. It contains the analysis of the imported resources and how much of the corresponding imports were not used by the application, measured in bytes. First a tabular view is provided, following by the corresponding graph.

TABLE 6.7: Table of the average imported and unused bytes values, collected via the Lighthouse tool in the prototype landscapes

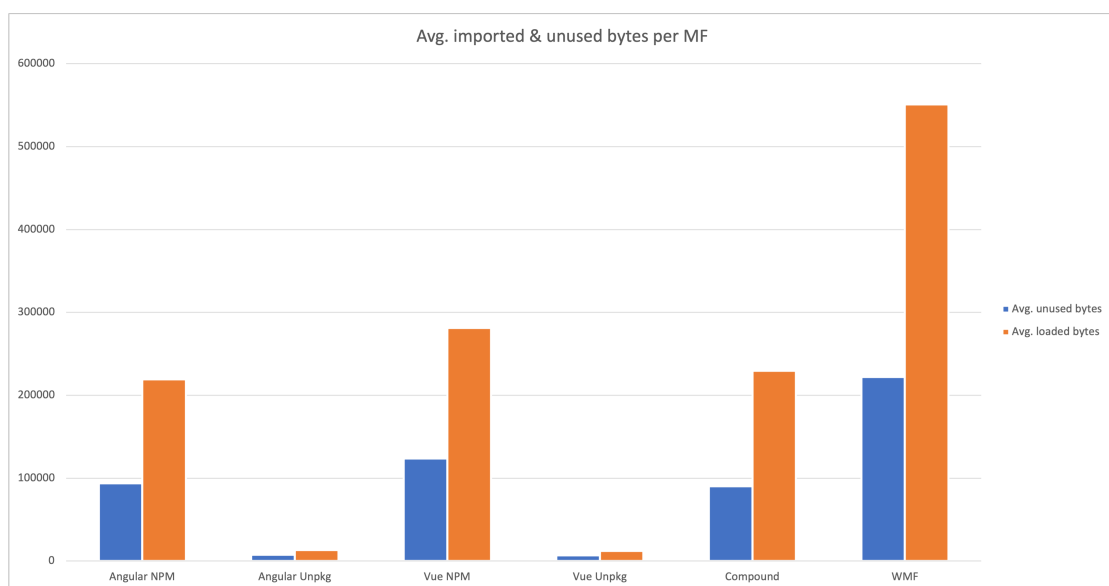| Landscape	Avg. imported bytes	Avg. unused bytes	Avg. unused bytes in %
Angular NPM	219257.14	93671.43	42.72
Angular Unpkg	13126.84	7425.87	56.57
Vue NPM	281262.33	123902	44.05
Vue Unpkg	12057.91	6844.21	56.76
Compound/ Web Components	229483.67	90265.19	39.33
Webpack Module Federation	551111.19	222113.73	40.31

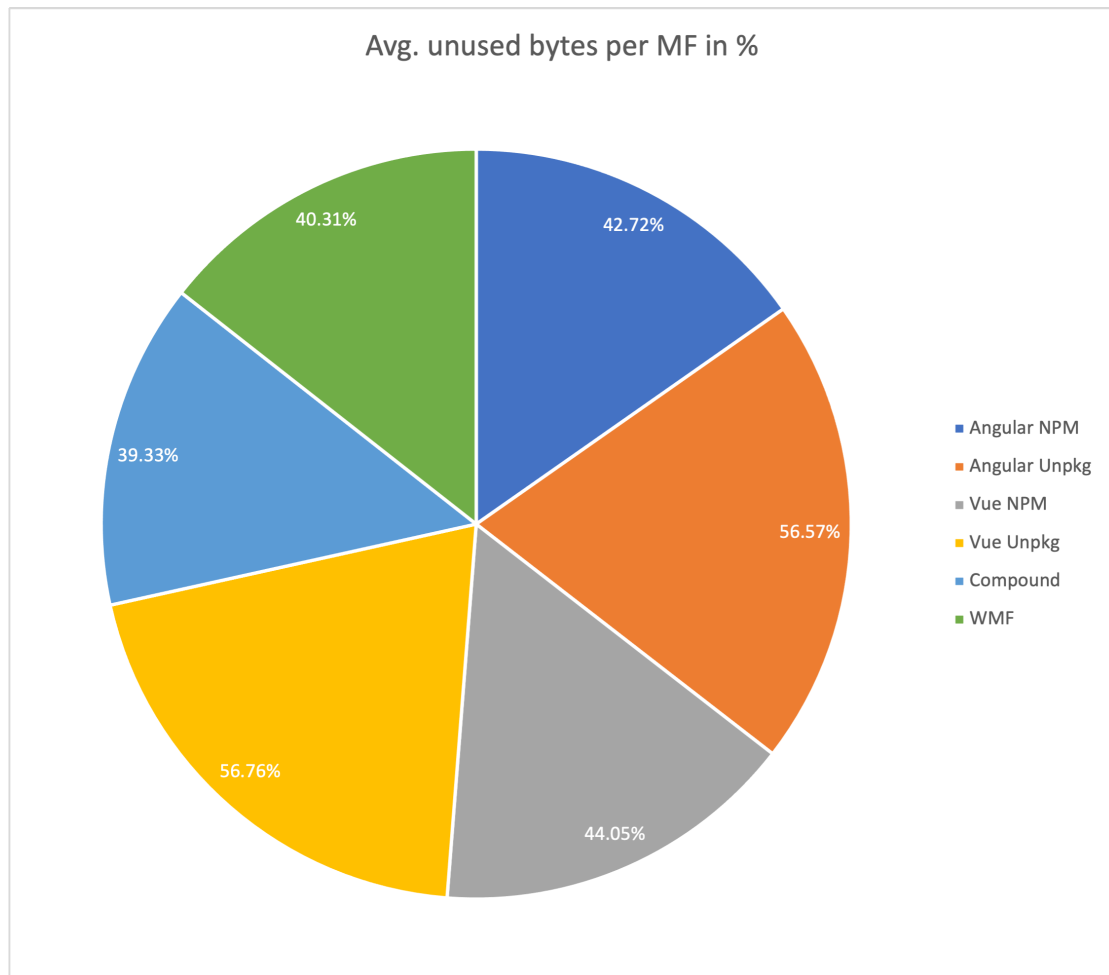FIGURE 6.2: Unused and imported bytes per landscape

FIGURE 6.3: Unused and imported bytes per landscape in %

The shown data in 6.7, 6.2 and 6.3 is highly application dependent. Still as these landscapes are considered to be generally representative, a display in efficiency can be drawn from the charts. The Unpkg CDN landscapes show a low unused bytes ratio. Since only the required resources were requested from the CDN, only a few unused bytes were detected by the Lighthouse tool. For the WMF landscape, it can be argued, that due to the bundling with Webpack, a lot of bytes were imported but only a few of them were not used. This can be explained through how the Module Federation works, since every configured shared resource is checked if it should be shared or not.

The following chapter will be concluding the displayed data. Detailed tables and information can be found in the appendixes A, B and C.

7. Conclusion

The following chapter will conclude and evaluate the results shown in chapter 6. First the a direct comparison between similar or equal landscapes and technologies is made, followed by the comparison of different landscapes and technologies. For the evaluation of the results, the introduced KPIs will be used as examples, references or for further calculations.

7.1 Direct evaluation of the landscapes

7.1.1 Angular and Vue landscapes

The Angular and Vue landscapes implemented were the following:

- Angular NPM 1
- Angular NPM 2
- Angular Unpkg 1
- Angular Unpkg 2
- Vue NPM 1
- Vue NPM 2
- Vue Unpkg 1
- Vue Unpkg 2

The naming refers to the technology used to load or access the required resources for the apps. Unpkg, as mentioned before, is a public cloud CDN. The idea behind those implementations is to prove that not only the CDN technology affects the performance of a micro frontend, but also the used frameworks. Comparing the numbers of those two landscapes it becomes obvious that, the Angular landscapes load more resources (counted by the URLs loaded KPI) in a shorter loading time for all implementations. Also looking at the initial loading times calculated via the loading procedure where caching was disabled, for all implementations Angular performed better compared to Vue.

It has to be mentioned, that the actual goal of this implementation was to compare the technology of the CDN, not the used frameworks. It still shows, that the framework has an affect on the performance of a micro frontends.

Following conclusions can be drawn by comparing the numbers of the NPM implementations and the ones with Unpkg in use:

1. The amount of requested URLs is significantly increased when CDN is used.

2. Even though approximately ten times the amount of URLs is requested by the CDN landscapes, the initial loading time decreases by 60 MS for the Angular Unpkg landscapes and for Vue it's even more.

3. The cache usage of the CDN landscapes significantly increases. This information is taken from the amount of *none established* ID occurrences. For each connection not established, a resource is loaded from the cache, thus the faster loading times, when cache is enabled.

4. A similar result is present in the loaded bytes from connection type. Compared to the NPM implementations, the Unpkg environments load significantly more bytes from *disk* or *memory*. This behavior was anticipated in the CDN implementations, since it is one of the desired features of this technologies. Every resource has a designated URL where it is loaded from, thus the browser can distinguish if it already present or not.

5. Additionally the average loaded content size of the Unpkg apps is lower, since only specific resources are requested from the CDN. Therefore big bundles with unused features are generally not present.

6. The graph 6.2 shows a similar result. Although the graph 6.3 shows, that approximately half of the imported bytes were not used according to the Lighthouse report, the absolute amount of loaded bytes is significantly lower, compared with the NPM environments.

7. When comparing the variances of the loading times, the Unpkg implementations show a lower value. Reasons for that could either be the protocol used by the CDN server or the smaller sizes of the loaded resources.

In retrospective, the KPIs **loading time in MS, resource sizes in bytes** and **amount of cached resources** the Unpkg implementations follow the expected patterns. Additionally looking at the variance, the Unpkg environments show a less variant loading time for all applications compared with the NPM implementations. One behavior was not expected though, it was assumed that the initial loading time of a CDN landscape would be significantly higher. Reason for that assumption was the effect of the network latency, since the resources are loaded from a remote server, instead of from an integrated bundle inside the project itself. Nonetheless, in case of the prototype this

behavior could not be confirmed, as even the initial loading times for the Unpkg apps were lower compared to the NPM apps. Reason for that result could be the efficiently picked resources. Instead of importing whole bundles of libraries only necessary components or resources were added as imports in the Unpkg landscape, thus the loaded byte size of those are so low.

Mentioning the *"efficiently picked resources"* another KPI which should be considered for all landscapes, is the effort connected to using a corresponding technology. This metric is hard to measure though, since it is highly influenced by the individual using the technology. A developer who is familiar with Webpack for instance, would have less trouble using and implementing the Module Federation. Therefore this metric is not easily quantifiable. Still, in the context of this thesis, the author will try and provide a subjective opinion on his experiences with the implementations he has done as generally as possible. In case of the CDN the effort of implementation was comparably low. From a developer's point of view it's even less effort, since no libraries have to be maintained in a central resource or package manager file (namely `package.json`). Nonetheless it has to be considered what type of CDN is used. In case of the prototype implementation, it was a public cloud CDN which already had all the required resources available. For a self-hosted CDN, this might not always be the case. Also when deciding to host an own CDN maintenance, development and deployment costs have to be put into account. This part was explained in chapter 2. In summary, the CDN technology is a comparably easy way to avoid redundancies in a micro frontend landscape. It still is connected to certain obstacles, when the use case is highly specific and requires certain customizations on CDN side.

Also multi-version landscapes are not supported by a CDN. That means that redundancies still can occur if the same resource is imported under different version tags. This use case is not directly covered by a CDN. If the resource itself has some sort of scoping feature a support can be provided (e.g. Custom Scopes by SAPUI5 Web Components). But this is not part of the CDN. Other technologies offer more support on that part.

7.1.2 Compound and WMF landscapes

The functionality of the Web Component and Module Federation landscapes to avoid redundant libraries were explained in chapters 4 and 5. Therefore, this section will focus on the direct comparison of these two environments. Since those two landscapes also include the aspect of heterogenic, multi-version micro frontends, this is considered in the comparison. Starting with values introduced in chapter 6, the first thing to attract attention should be the difference in the average loading times of the landscapes. For

the caching disabled scenarios, the WMF landscape requires almost a fifth of the time compared to the compound landscapes. With caching enabled the difference is only a third but still significant. Additionally the bytes loaded by the landscapes differ. WMF loads almost double the amount of bytes compared to the compound environments in a shorter period on average. Additionally WMF does this with barely using the cache, even with caching enabled. Where the compound environment loaded approximately 756055 bytes on average from memory, the WMF only imported 1959 bytes on average from the cache. Therefore, based in this comparison the Webpack Module Federation seems to be on top.

Still further aspects have to be considered when using those technologies. The first is the multi-version handling. Web Components offer means to register different versions of a component, by adding e.g. a suffix to its tag name. In case of the prototype a component library provided such a feature. Still a self-developed component can implement a similar functionality. This again would be connected to more effort developing Web Components for a compound landscape in Luigi, but it is not entirely impossible. When using an existing component library, this feature might already be present as it is the case for the SAPUI5 Web Components. Since each other version of a component is registered under a different tag, with a version suffix attached to it, the result might lead to redundancies again. That means a component called e.g. `ui5-table` would be registered and imported twice into the same landscape under different names. For instance version 1 would be `ui5-table-v1` and version 2 `ui5-table-v2`. Thus even though the different versions are handled in a distinguishable way, the redundancies would increase.

The Module Federation offers different means for solving this issue. By sharing certain dependencies and with a definition of a required version, redundancies are not entirely eliminated in a multi version WMF landscape, but handled more elegantly. During the data collection of this landscape, a phenomenon appeared which seems to be intended by the Module Federation's developers. Shared dependencies like for instance `@angular-common - v1.1.2`, are always loaded via the network even if caching is enabled, but never in full size. On initial loading this dependency would be approximately 1.2 MB in size, but when reloading the page with cache enabled, it is loaded again but with 412 KB in size. This behavior appeared for all shared dependencies in the WMF landscape. It is enabled via the Module Federation itself, since the shared dependencies are lazy loaded as chunks.[34]

Again it seems to be that the WMF is superior on that regard compared to Web Components. Still following aspect is not considered yet, the effort of using the technologies. As mentioned in the previous section this is no actual KPI but rather a subjective opinion of the author. When directly compared, the effort of implementing or developing micro

frontends, was by far higher when using WMF compared to Web Components. Even taking into account that a component library was used, still the Module Federation required more expertise with the Webpack bundler. The possibilities with WMF are versatile, but to a layman unfamiliar with the necessary bundler Webpack, hard to use. The documentation offers good hints and explanations for options and syntax inside the configuration, but it is documented in a more general way and if the developer wants to use it in combination with a Webpack based framework like Angular or Vue, the whole operation becomes more experimental. In the case of Angular, a separate dependency is required to publish the hidden `webpack.config.js` via which the Module Federation is enabled in the first place. And this configuration has to be done for every module or remote, federated in the landscape. Applied to a real life scenario this might become a bigger obstacle, as it was in case of the prototype. Independent, isolated teams would work on the same micro frontend landscape, using different UI frameworks for their remote modules. Not only the bundle sizes of the heterogenic landscape would increase, since every shared dependency has to be bundled and published in the landscape, but the routing inside the landscape becomes a challenge itself. In case of Angular for instance, the inner routers do not recognize route changes. One router would have to import another application's router manually, in order to be able to communicate route changes.[34]

Web Components on the other hand are based on standards. The result and assigning process is the same. Different UI frameworks offer means to register developed components as Web Components (e.g. Angular Elements). Thus the developer can work framework-independent within a familiar environment and expect the same result as another developer working with the tools of his choice. Additionally Web Components are not bound to certain technology stacks - unlike the WMF which requires the Webpack bundler to enable it. Therefore, Web Components have less limitations and more stability due to their standardization.

In summary, both technologies offer means to solve the issue of avoiding redundant libraries, but when it comes to handle multi-version landscapes, WMF offers more elegant ways compared to Web Components. On the other hand using the WMF limits the developer to a certain technology stack and is not always easy or effortless implemented depending on the UI framework in use. Also certain obstacles are present in the WMF as of now and require workarounds to solve them.[34] On that regard Web Components offer easier means and ways for implementation due to their standardized aspects.

7.2 Final recommendations

The previous sections contained general summaries for each respective technology, based on either empiric data, official sources or subjective experience. The following listing will provide final recommendations when and how best to use the introduced and implemented technologies, based on the conclusions made in this transcript.

- **Content Delivery Networks** - CDN offer an easy way to reduce redundancies in micro frontend landscapes by centralizing the landscapes resources to one point. From a developer-perspective only the way of importing the resources changes. Therefore, this is the easiest and fastest way of achieving the goal of avoiding redundant libraries in micro frontend landscapes. Nonetheless, what it offers in simplicity it lacks in flexibility. Multi-version support is not always present and can not be entirely solved by this technology. Also a public CDN might not always have the necessary resources required by the landscape and hosting an own CDN is connected to high maintenance and developing costs (depending on its size). Therefore, if the use case describes a homogenic micro frontend landscape, one without different versions, CDN is the way to go. If different versions of the same resource are required, there are more elegant ways than this technology.

- **Web Components** - As a web standard it is a save way of avoiding redundancies, maybe not in libraries but rather in the used components themselves. By providing reusable components to the browser which are not affected by the isolation of micro frontends, this technology reduces the amount of used components in the landscape. Additionally existing Web Component libraries offer means to scope different versions of its components, providing a way to handle multi-versions inside the landscape. With its standardized aspect it also does not limit the developer to certain technology stacks and is compatible with most common UI frameworks. Some even offer framework features to create Web Components from their projects like Angular Elements. If the use case requires a lot of reusable components with as less redundancies as possible, Web Components offer the best way to provide that service.

- **Webpack Module Federation** - A rather new technology compared to the other two, promising to excel where the other two lack. It can federalize any piece of precompiled code and serve it to the landscape. This module or remote is then embedded into a host application. The remote itself can either be a UI component, a module or a utility service, thus providing a flexible way of sharing code inside its landscapes. The possibilities are vast using this technology. But this offer comes

for a price: One has to use the Webpack bundler and for certain features or issues workarounds are required. Subsequently it can be said, this technology, if used correctly, is applicable to almost any use case. But it also might be connected with some obstacles and workarounds, which could be easier solved with other means.

As a closing word for the recommendation: When the main goal is to avoid redundant libraries in a micro frontend landscape, each of the introduced technologies offer means to do that. But each comes with their own kind of trade-off. Also when picking from one of the above choices, side effects and benefits have to be considered. Therefore, the final recommendation is highly dependent on the use case and requirements for the landscape to be developed. Additionally it is not excluded or impossible to combine those technologies.

8. Prospect

The time available for the research, data collection and the actual writing was limited and therefore this transcript is scoped to a certain degree. Nonetheless the statements in this document are not final and further possibilities can be explored in that field. One of which was mentioned in the last chapter: The combinational effect of the researched technologies. For instance, a Web Component based micro frontend landscape, in which the resources for the components are provided by a CDN. Also the WMF topic was analyzed in the context of the UI framework Angular. Even though it is a valid way of doing, the Module Federation can be used in combination with other frameworks too. This is definitely a field which should be looked into. Especially when taking into account that it was shown, that a framework affects the performance of a micro frontend. Another field which was not dealt with, is the development of an own CDN. Even though a approximate assumption was made concerning the effort connected to such a project, this is by no means empiric data. Therefore since the developed landscape rely on the Unpkg API to request the CDN resources, it would be an interesting experiment if and how an own CDN could improve or optimize the performance metrics for similar landscapes. Lastly the Surge web server, used for the deployment of the landscapes, was connected to certain limitations too, namely the missing HTTP/2 server configuration. Thus, it would make sense to deploy those landscapes over different web servers just to see if the changes on server-side improve the performances in the given context.

It is save to say, the implemented prototype can be considered to be representative but still offers room for improvement and optimization. Thus, when the given research is applied to a real life scenario, special conditions or requirements have to be considered when making a decision in that context. Therefore, it is mentioned in chapter 7 that the gain or benefit of each respective technology is highly dependent on the individual use case.

Bibliography

[1] Angular CLI - ng build. https://angular.io/cli/build, February 2021. Accessed: 16.02.2022.

[2] Chiradeep BasuMallick. Top 10 Content Delivery Network (CDN) Providers in 2021. https://www.toolbox.com/tech/networking/articles/content-delivery-network-providers/, August 2021. Accessed: 28.01.2022.

[3] Eric Bidelman. Custom Elements v1: Reusable Web Components. https://developers.google.com/web/fundamentals/web-components/customelements, August 2020. Accessed: 19.01.2022.

[4] Eric Bidelman. Shadow DOM v1: Self-Contained Web Components. https://developers.google.com/web/fundamentals/web-components/shadowdom, July 2020. Accessed: 02.02.2022.

[5] Vue - Build Targets. https://cli.vuejs.org/guide/build-targets.html#library, November 2019. Accessed: 16.02.2022.

[6] MDN contributors. Using shadow DOM. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM, October 2021. Accessed: 02.02.2022.

[7] MDN contributors. Using templates and slots. https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_templates_and_slots, October 2021. Accessed: 07.02.2022.

[8] MDN Web Docs. Web APIs - CustomElementRegistry. https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry/define, October 2021. Accessed: 19.01.2022.

[9] MDN Web Docs. JavaScript modules. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules, January 2022. Accessed: 25.01.2022.

[10] MDN Web Docs. Web Components. https://developer.mozilla.org/en-US/docs/Web/Web_Components, January 2022. Accessed: 19.01.2022.

[11] UI5 Web Components documentation. UI5 Web Components Scoping feature. https://sap.github.io/ui5-webcomponents/playground/advanced/scoping/. Accessed: 19.01.2022.

[12] Michael Geers. Micro Frontends extending the microservice idea to frontend development. https://micro-frontends.org/. Accessed: 10.03.2022.

[13] How to enable HTTP/2 protocol on the website. https://serpstat.com/blog/how-to-enable-http2-protocol-on-the-website/. Accessed: 17.02.2022.

[14] imperva.com. CDN and SSL/TLS. https://www.imperva.com/learn/performance/cdn-and-ssl-tls/. Accessed: 01.02.2022.

[15] imperva.com. CDN Architecture. https://www.imperva.com/learn/performance/cdn-architecture/. Accessed: 31.01.2022.

[16] imperva.com. CDN Route Optimization. https://www.imperva.com/learn/performance/route-optimization-anycast/. Accessed: 01.02.2022.

[17] imperva.com. Front End Optimization. https://www.imperva.com/learn/performance/front-end-optimization-feo/. Accessed: 01.02.2022.

[18] imperva.com. Origin Server. https://www.imperva.com/learn/performance/origin-server/#:~:text=A%20CDN%20topology%20distinguishes%20between,of%20some%20of%20its%20content. Accessed: 31.01.2022.

[19] imperva.com. What is a CDN. https://www.imperva.com/learn/performance/what-is-cdn-how-it-works/. Accessed: 31.01.2022.

[20] Lifecycle event sequence. https://angular.io/guide/lifecycle-hooks#lifecycle-event-sequence, September 2021. Accessed: 16.02.2022.

[21] Project "Luigi". Node parameters. https://docs.luigi-project.io/docs/navigation-parameters-reference. Accessed: 10.02.2022.

[22] Project "Luigi". Web Component.
https://docs.luigi-project.io/docs/web-component. Accessed: 10.02.2022.

[23] Project "Luigi". Luigi architecture.
https://docs.luigi-project.io/docs/luigi-architecture, October 2019.
Accessed: 17.01.2022.

[24] Project "Luigi". Luigi Client API documentation.
https://docs.luigi-project.io/docs/luigi-client-api, October 2019.
Accessed: 17.01.2022.

[25] Project "Luigi". Luigi Core API documentation.
https://docs.luigi-project.io/docs/luigi-core-api, October 2019.
Accessed: 17.01.2022.

[26] Project "Luigi". Luigi documentation overview.
https://docs.luigi-project.io/docs/getting-started, October 2019.
Accessed: 17.01.2022.

[27] Manuel Martín. Open-WC Scoping. https:
//dev.to/open-wc/the-evolution-of-open-wc-scoped-elements-195b, April
2020. Accessed: 19.01.2022.

[28] Severi Peltonen, Luca Mezzalira, and Davide Taibi. Motivations, benefits, and
issues for adopting micro-frontends: A multivocal literature review. *Information
and Software Technology*, 136:106571, 2021.

[29] Reacht - Bundling.
https://reactjs.org/docs/code-splitting.html#bundling, February 2022.
Accessed: 16.02.2022.

[30] Simon Rey. Conception and prototypical implementation of a web components
based library in the context of reusability. page 8–10, 2021.

[31] Manfred Steyer. Dynamic Module Federation with Angular.
https://www.angulararchitects.io/aktuelles/dynamic-module-federation-
with-angular/, June 2020. Accessed: 16.02.2022.

[32] Manfred Steyer. Getting Out of Version-Mismatch-Hell with Module Federation.
https://www.angulararchitects.io/aktuelles/getting-out-of-version-
mismatch-hell-with-module-federation/, September 2020. Accessed:
21.01.2022.

[33] Manfred Steyer. Multi-Framework and -Version Micro Frontends with Module Federation: The Good, the Bad, the Ugly. https://www.angulararchitects.io/aktuelles/multi-framework-and-version-micro-frontends-with-module-federation-the-good-the-bad-the-ugly/, June 2020. Accessed: 21.01.2022.

[34] Manfred Steyer. Multi-Framework and -Version Micro Frontends with Module Federation: The Good, the Bad, the Ugly. https://www.angulararchitects.io/en/aktuelles/multi-framework-and-version-micro-frontends-with-module-federation-the-good-the-bad-the-ugly/, June 2020. Accessed: 28.02.2022.

[35] Manfred Steyer. The Microfrontend Revolution: Module Federation in Webpack 5. https://www.angulararchitects.io/aktuelles/the-microfrontend-revolution-module-federation-in-webpack-5/, April 2020. Accessed: 21.01.2022.

[36] Manfred Steyer. The Microfrontend Revolution: Module Federation with Angular. https://www.angulararchitects.io/aktuelles/the-microfrontend-revolution-part-2-module-federation-with-angular/, April 2020. Accessed: 16.02.2022.

[37] Todor Stoyanov. Github - UI5 Web Components. https://github.com/SAP/ui5-webcomponents. Accessed: 10.02.2022.

[38] Google Trends. Google Trends. https://trends.google.com/trends/explore?date=2018-01-01%202022-01-11&q=micro%20frontend,microfrontend, January 2022. Accessed: 11.01.2022.

[39] Sipat Triukose, Zhihua Wen, and Michael Rabinovich. Measuring a commercial content delivery network. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 467–476, New York, NY, USA, 2011. Association for Computing Machinery.

[40] unpkg.com Documentation. https://unpkg.com/. Accessed: 25.01.2022.

[41] Tapan Vora. Top 10 micro frontend frameworks to consider. https://www.cuelogic.com/blog/micro-frontends-frameworks, December 2020. Accessed: 18.01.2022.

[42] webcomponents.org. Specifications - The ES Module specification. https://www.webcomponents.org/specs#the-es-module-specification. Accessed: 07.02.2022.

[43] Webcomponents.org. Web Components Best Practices. https://www.webcomponents.org/community/articles/web-components-best-practices, April 2014. Accessed: 19.01.2022.

[44] Caleb Williams. Encapsulating Style and Structure with Shadow DOM. https://css-tricks.com/encapsulating-style-and-structure-with-shadow-dom/, March 2019. Accessed: 10.03.2022.

[45] Caifang Yang, Chuanchang Liu, and Zhiyuan Su. Research and application of micro frontends. *IOP Conference Series: Materials Science and Engineering*, 490:062082, April 2019.

[46] Antonello Zanini. 5 Reasons You Should Adopt a Micro Frontend Architecture. https://www.sitepoint.com/micro-frontend-architecture-benefits/, December 2021. Accessed: 06.03.2022.

List of Figures

Glossary

API **A**pplication **P**rogramming **I**nterface

A. Results for all prototype landscapes

B.   Lighthouse result table

C. Lighthouse result graphs