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.

Karlsruhe, den 28. 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 each of the respective technologies has their own means to avoid redundant libraries in the micro frontend landscapes, be it either distinguishable resource URLs provided by the CDN, reusable Web Components or even shared dependency through the Module Federation. Each technology has its own approach of solving the issue at hand. Still, every method has limitation which are evaluated as well, such as the effort involved when using one of the corresponding technologies.

For CDN, it is shown that even though it is easy to implement and use in the project itself, a public CDN has certain restrictions. These include the absence of special libraries required for the project and the constant dependency on the availability of the CDN servers. Hosting your 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 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 all technologies provide 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.[1] 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 etc.[2] But still, this concept has certain flaws which have to be considered when making the decision.[3]

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, since 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.[4][5]

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

1.2 Definition of redundancies

For the context of this transcript, the term *"redundancies"* or *"redundant libraries"* refers to 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 subsequent chapters.

1.3 Goal

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

To achieve this, two prototypes are implemented using the Luigi micro frontend framework. The landscape itself is designed to be as representative as possible by using a heterogenic tech stack. Additionally following methods are present to avoid redundantly loaded libraries:

- Content Delivery Network (CDN)
- Web Components in Luigi Compound views
- Webpack Module Federation (WMF) framework

The implemented methods are evaluated under predefined metrics. The steps to collect the data for the metrics are explained too. Eventually a conclusion is drawn based on the empirical data, accompanied by the subjective opinion of the author. It is explained how the usage and implementation of the respective technologies differs and which advantages each technology brings to the table.

1.4 Scope of Work

To account for the limited time frame for this transcript, the following aspects are either not considered or referenced only on a theoretical level.

- The development of a self-hosted 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 those necessary for the WMF context.
- Configuration for the Luigi framework of the implemented landscape - Each implemented prototype uses this framework for its micro frontend landscapes. 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.

1.5 The collaborators

This thesis was written in collaboration with SAP Luigi project team. The SAP was 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 was originally founded in 1997, was later acquired by SAP in 2013, after moving its headquarter from Zug in Switzerland to Munich in Germany.[6] 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. Micro frontend framework Luigi

In this chapter, an overview about the used Luigi framework is given.

Currently there are different micro frontend frameworks available on the market, including but not limited to **Bit, SystemJS, Webpack 5 Module Federation, Piral, Single SPA** and **Luigi**.[7] For the implementation of the representative landscapes the Luigi framework was used. Therefore, a short introduction to Luigi it is given here.

2.1 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.[8]

2.1.1 Core

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

- Navigation - enables navigation between micro frontends
- Authorization - enables authorization for the landscape
- Localization - providing translations to display its applications in multiple languages
- General settings - e.g. header display configurations, enable 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 listing 2.1.[1]

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

```
- react-core-mf
  - [...]
  - node_modules
  - public
    - [...]
    - index.html
    - luigi.config.js
  - [...]
  - 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.[9]

2.1.2 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 the Client. 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 straightforward approach is the direct import with a HTML script tag. Another option would be the import of the local package manager dependency.[10]

```
<!-- Via a HTML script tag -->
<script src="https://unpkg.com/@luigi-project/client@1.17.0/
 luigi-client.js">
</script>

<!-- Via a package manager -->
import LuigiClient from "@luigi-project/client";
```

LISTING 2.2: Import methods of the Luigi Client

2.2 Architecture

After the short introduction of the framework's core components, a general architecture of a landscape using Luigi is provided in figure 2.1.

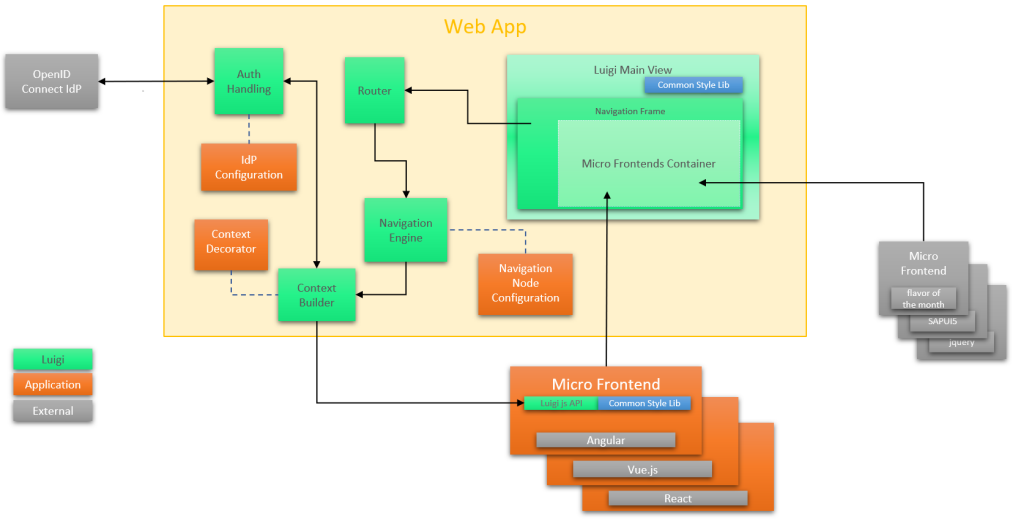

FIGURE 2.1: Architecture of the Luigi Framework [11]

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 (Navigation Frame), 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 are the redundantly imported libraries of the embedded micro frontends. The reason for this problem is that each node is an isolated application. For instance, navigating to the first micro frontend, 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. The reason for that is the bundling of projects which makes the references indistinguishable to the browser. The following chapters will introduce technologies, which offer methods to resolve this issue.

3. Content Delivery Networks

One way of avoiding redundant libraries 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). 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.[12]

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 these 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 not necessarily 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.[13]

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 need 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. However costs attached to this scenario can be immense and exclude this method due to cost inefficiency.[14]

This chapter will give further information about the CDN technology, how it functions and what further features it offers to optimize a micro frontends or websites performance.

3.1 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.
- **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 improves 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**. 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.[15]

An example for a basic CDN distribution is shown in figure 3.1.

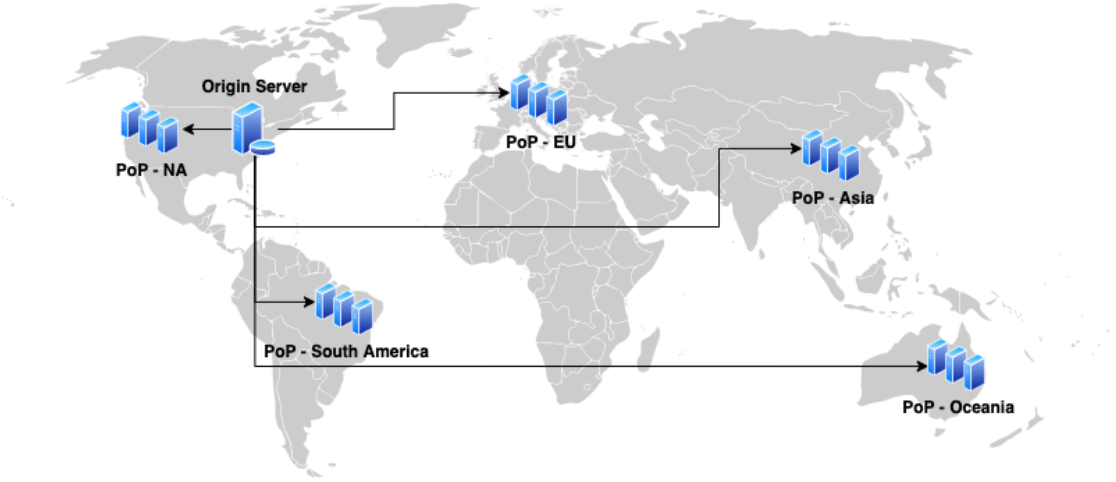

FIGURE 3.1: Basic distribution a CDN

It has to be mentioned that the geographical distribution in figure 3.1 is simplified, for the sake of readability. In a real scenario, the cache servers of each PoP would themselves be distributed over different data centers on the continent. A client requesting resources from the CDN always communicates with the nearest cache server to their designated location.[13]

3.2 Design

For a CDN to fulfill its purpose, four requirements have to be met. 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:

- **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 a website, the negative effect on user experience can cause financial harm to the host.[13]
- **Scalability** - Since a 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 cannot be met which 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 to its PoPs over the globe accordingly. 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.[15]

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

- **The scattered CDN** - This topology focuses on physical proximity. The PoPs are kept rather small in size, but are scattered around the world in greater frequency, thus providing as much proximity to their client as possible. This topology excels in providing the CDN's 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 accrues more maintenance costs, since more PoPs are required to be maintained. Also, deploying new configurations can be connected to a lot of

effort, depending on the number of PoPs scattered across the CDN. Additionally, this number can also affect the RTT since every PoP inbetween the client and the server is a connection point.

- **The consolidated CDN** - Contrary to 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 servers available to the PoPs are highly advanced and provide a lot of hardware capacity. Additionally, since the number of major data centers around the world is rather limited, the number of PoPs is reduced as well, compared to the scattered topology. Following the quality over quantity principle, a PoP in this topology can handle a greater amount of traffic compared to its counterpart, and is also more resilient, specifically when it comes to DDoS attacks. Also, due to the moderate number 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 number of requests, its reach to low-connectivity regions is rather limited. This is due to the proximity difference between the servers and clients. Additionally, deploying a new PoP into this topology requires 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 are intended to solve specific issues or challenges. [16]

3.3 Optimization

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

3.3.1 Route optimization with Anycast

As previously mentioned, a CDN can be designed with different topologies which can affect its performance. Another factor which has to be considered in this equation is the routing itself. Basically, it does not matter if a cache server is located in close proximity to a client, if the requested resource is not present there. In this case, the client would have to be routed to different, cache servers located further away. To circumvent this, 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 a 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 number of time a request changes hands between hosts.[17]

3.3.2 TLS Performance

The route optimization with Anycast, also improves the RTT when using the TLS/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, TLS (Transport Layer Security) is a protocol via which secure communications are ensured on the Internet. The communicating parties establish a connection via the 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. Symmetric keys for encoding and decoding are generated

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

A CDN can provide improvement to this overhead and decrease the RTT of a request. Through the aspect of route optimization and 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/TLS negotiation process is shorter, too.[18]

3.3.3 Frontend optimization

The term frontend optimization refers to the process of making a website more browser-friendly and reducing loading times. There are multiple ways 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 number of which is actually limited by 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 by pre-pooling connections and ensuring 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 introduces the method of multiplexing. This allows a single TCP connection to transfer multiple different HTTP requests [19].

- **File compression** - Of course it is not always about the number of requests per site 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 close by. 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 the cache of a nearby cache server. Locally stored static files do not have to be loaded via the network and are available to the browser for rendering almost immediately. The only question remaining is how long does a resource have to be cached. This information is necessary to optimize the use of the client's 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 have also started to use machine learning techniques to follow and understand content usage patterns and automatically optimize caching policies.

- **Code minification** - Similar to the file compression method, the process of code minification offers a way of reducing file sizes too. Whereas a developer writes code in a humanly readable way, with spaces and line breaks, a machine does not need this kind of formatting. By removing comments, spaces and line breaks, the size of a code file can be reduced by 30%. CDNs use methods like gzip, minify or a combination of these two 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 to reduce the actual size of the image and thus the loading time. However, 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 quickly and then progressively replace it with higher-resolution variants. Alternatively, a website host could use vector or raster images. These are resolution independent, smaller in size and highly responsive.[20]

These methods in combination with the CDN technology provide a possible gain in user experience for a website host. [21]

3.4 Hosting 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.

This topic will be showcased based on a hypothetical scenario. The numbers and metrics of this scenario are either assumed or taken directly from the implemented prototype.

The first requirements of the scenario are the following:

- 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 same team and are only available for internal use.

These circumstances ensure that only a self-hosted CDN is applicable. Next, the numbers and further requirements of the landscape are necessary. Those are required for the cost calculation of the CDN. It is mentioned if the numbers are actually *assumed/estimated* or *taken from the prototype.*

- How often the site is accessed per day - 50000 times (estimated)
- Avg. byte size per page load without caching on client side - 8983.5 KB (taken from the prototype)

- Avg. amount of GET requests to CDN per site load - 198 requests (taken from the prototype)
- The site is only accessed in North America

Based on those requirements, the following values are calculated for the monthly CDN usage of the described scenario 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:

$$B = 50000 \times 30 \times 8983.5 = 13475250000\,\text{KB} = 13.47525\,\text{TB}$$

Next, it will be calculated how much it would cost a company or department to pay for the CDN defined in this scenario.

There are multiple CDN solution providers on the market, including **Amazon CloudFront, Azure CDN, Google Cloud CDN**. [22] Since the pricing models of the mentioned providers differ, it is difficult 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
- Google Cloud CDN - 1168.99$ per month
- Azure CDN - 1123.90$ per month

After averaging those values an *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, which 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 PoPs around the world. Also, locations play a key role, as different regions pay more for traffic then others and this value is provider-specific. Since it is not the goal to pick a provider here, but rather to provide a general overview of costs for such a solution, those additional features were not considered for the given scenario.

4. Web Components

Another option to address the issue of redundancies are Web Components. Consisting of the following four standards, they provide reusable elements encapsulated in HTML tags.[23]

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

The following sections will introduce the four standards of Web Components with their functions and showcase what purpose they can serve for the context of this thesis.

4.1 Custom Elements

This standard of Web Components provides an API via which new HTML tags can be defined and registered by the `CustomElementRegistry`. Since a single tag name can only be registered once, multiple registrations of the same element would lead to an exception, thereby making already registered tags reusable for the whole micro frontend landscape without reloading the code of the registered element.[24]

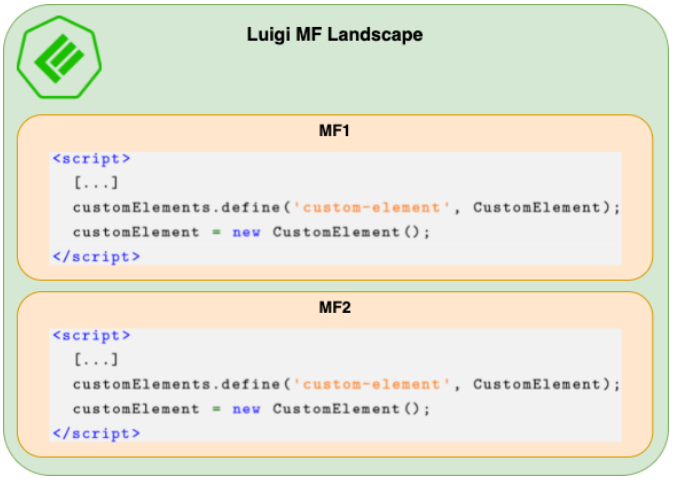

FIGURE 4.1: Simple micro frontend landscape using Web Components

Figure 4.1 shows a simple micro frontend landscape with Luigi using Web Components. As displayed, 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 scenario shown will cause a `DOMException`, 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 cannot be overwritten without reloading the page.[25] Therefore, this standard is crucial for the goal to achieve in this transcript, by making impossible to register same components with the same tag name.

4.2 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. [26] 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.[27]

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 4.2 visualizes the relations between the newly introduced terms.

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

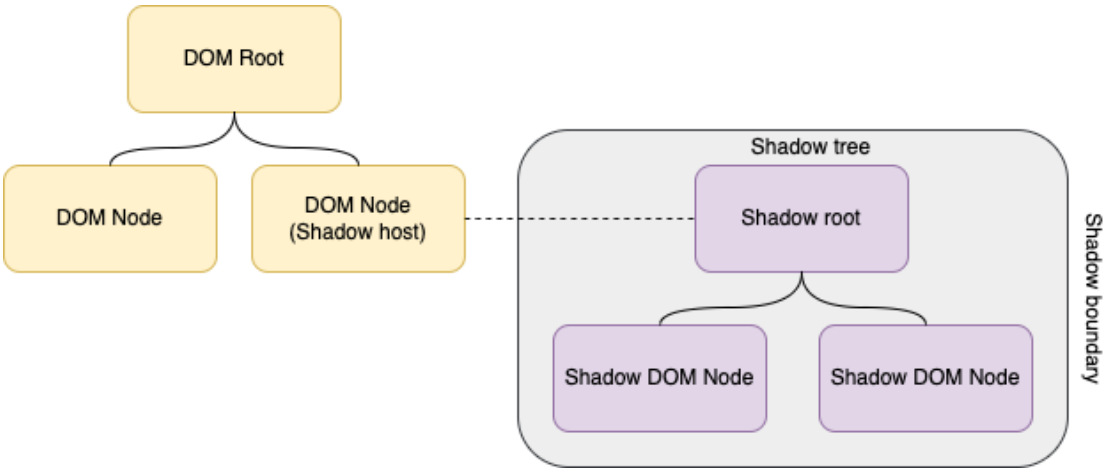

FIGURE 4.2: Shadow DOM architecture

and other configurations applied to a tag inside the Shadow DOM are not applied to the actual DOM.

The Shadow DOM, even though separated from the DOM, has an attribute via which it can be made accessible. By adding the `mode` attribute to the `this.attachShadow()` method, the visibility of the Shadow DOM, to the document can be defined [27]. Possible values for the `mode` property are:

- **open**
- **closed**

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`.[28] 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.[29]

For the context of this thesis, the Shadow DOM standard offers means to individualize the registered Web Components of a landscape. It can be used to apply customizations or add individual event listeners to specific components. This comes with a trade-off though. The level of isolation increases through the encapsulation of code elements between the single components. This increase makes it difficult to 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.

4.3 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 4.1 and **??** provide examples how exactly these two standards are used. [30]

```
<!-- 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 4.1: Definition and usage of the `template` standard [30]

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 JavaScript.

```
<!-- Definition of the slot -->
<p>
  <slot name="my-text">Default input</slot>
</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 4.2: Definition and usage of the `slot` standard [30]

As it can be seen in listing 4.2, 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.[30]

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. Similar to the Shadow DOM this feature could be used to apply more flexibility to the Web Components, by adding placeholders for customization. But contrary to the Shadow DOM, this does not increase the level of isolation, since the rendered elements are not separated from the actual DOM. Redundant code snippets can be defined as reusable templates, thereby reducing the redundancies in the code itself.

4.4 ES Modules

The last standard is not referred to by every source, but according to [31], 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 4.3: Importing modular Java Script documents into another [31]

Listing 4.3 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.[31]

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 **??**. Therefore, this aspect enables a component to be imported in multiple micro frontends as a module.

4.5 Additional considerations

This chapter has shown, that this technology solves the issue of redundancies in a micro frontend landscape, on a different level. Instead of reducing the redundant libraries, it actually reduces the code itself used by the landscape.

There is still a niche case which can be raised as an objection though. Since the registration and therefore the naming of elements is up to the developers, it can occur that two components are registered under the same tag name, but the elements themselves are different. In this case, 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 namespace for every team when creating Web Components e.g. Team 1 has the namespace `team-1`, causing a Web Component registered by the micro frontend of Team 1 to be named `team-1-tagname`. This 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.[32]

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 a set of unique registrable tags. If another micro frontend would try and register 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 is employed, when versions of the used components differ between micro frontends. It 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. [33] [34]

5. Webpack Module Federation

A rather new technology on the market is the **Webpack 5 Module Federation (WMF)**. The Module Federation was part of Webpack's 5th version, released on October 10, 2020. It added features which improved its usage for developing micro frontends.[35] The way it does that, is via modularizing self-compiled code parts and publishing them for integration by other modules. This published modules 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 **remotes** 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] The usage of the WMF is tied to the Webpack bundler, since the necessary configuration is done in the `webpack.config.js`. This restriction is applied to every **remote** in the WMF landscape, not only to the `host`.

WMF provides means to solve the issue, described in this thesis. First a short introduction of WMF is given, then the shared dependency feature is explained. It is mentioned, that 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.

5.1 Enabling the Module Federation

Prior to introducing the usage of the Module Federation, it is necessary to introduce Webpack itself first, as it is a mandatory feature for using the Module Federation. The documentations of popular UI frameworks, like React, VueJS or Angular, imply that Webpack is used by default, but this can customized by a developer is needed [36][37][38].

To change the default configurations, the necessary `webpack.config.js` file has to be enabled first. In case of Angular, two dependencies have to be installed, using e.g. the Angular CLI. By enabling the Webpack configuration file, the features of the Module Federation are enabled, too. 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

These commands are necessary, since the CLI protects the Webpack configuration from access. The `@angular-architects/module-federation` package provides a custom builder, via which this access restriction is lifted. After installing this dependency in an Angular project, a `webpack.config.js` will appear on root level of the corresponding project.[39] This dependency has to be added in each **remote** or **host** of the WMF landscape, to enable the Module Federation in those projects.

After the enabling, the necessary configuration can be applied to the `webpack.config.js` file. The **remotes** publish their modules and **hosts** consume them.

5.2 Shared dependency feature

The shared dependency feature of WMF, is a way to solve the issue of redundant libraries in its landscapes. 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. Another option can be, a less restricted dependency definition, which results in the framework, loading of highest major version available. WMF is able to distinguish between major versions of the dependencies, shared in its landscapes. If not configured otherwise, it automatically picks the highest major version and applies it to the sharing modules.

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

Listing 5.2 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**.[40]
- `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 does not apply one dependency to all remotes if different major releases are configured for them. If a higher version of the same major release is available, it will only be loaded for the modules with the same major release configured. Modules with a lower or higher, major release required, are not affected by it [41]. This aspect is explained further below.

Now, when it comes to sharing the dependencies inside the micro frontend landscape, each **remote** has to participate. 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 taken from 5.2, it becomes obvious that multiple versions of the same framework can exist in a landscape.

5.3 Multi-version landscapes in WMF

Figure 5.1 illustrates the case, of how WMF handles a multi-version landscape.

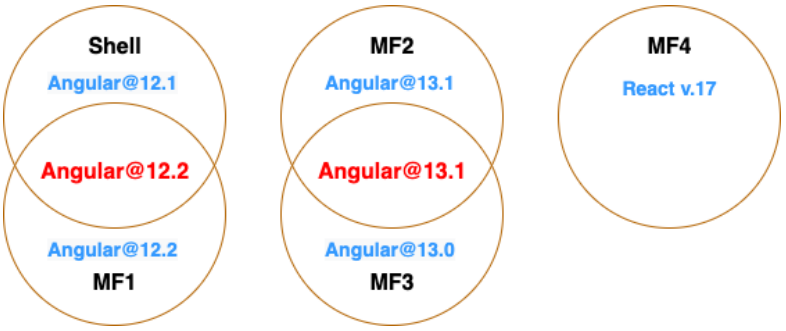

FIGURE 5.1: WMF way of handling multi-versions

As it can be seen, WMF always picks the latest major release, assuming the respective micro frontends have a similar configuration as shown in listing 5.2 applied. If, for instance, **MF3** has the `strictVersion` property set to `true`, for a dependency, it would cause the loading this library, too. Even if, it is already present in a different version.

There are side effects to sharing the same dependency over the whole micro frontend landscape. One of which is the increase in bundle sizes, since every **remote** bundles its local dependencies. WMF then, picks the ones to serve during the runtime of the landscape. This impact has a trade-off tough. Returning users can benefit from cached dependencies, since the same bundles are reused, loading the site.[41]

Therefore, WMF is able to reduce redundant libraries in micro frontend landscapes and additionally supports multi-version environments, via the shared dependency feature.

6. Development of the prototypes

This chapter will provide an overview of the prototype landscapes and explain how the previously introduced technologies were used in them.

6.1 Definition of representation

Before the implementation is showcased, it is explained why and how representation is defined for the context of the thesis. Since not every micro frontend landscape is the same, a general definition was required before developing the prototypes. For clarification, the term prototype is a synonym for landscape and vise versa, in the following context. Those prototypes had to fulfill most of the following requirements, to be considered representative:

1. The landscape has to contain at least 6 or more micro frontends
2. The different micro frontends in the landscape, have to load the same libraries/dependencies to cause redundancies
3. The tech stack of the landscape has to be heterogenic (not only one UI framework is used).
4. The embedded micro frontends have to be isolated projects
5. The landscape contains different versions of a dependency

In specific cases, not all requirements could be met, it is explained why though.

6.2 Prototype overviews

This section will shortly describe the developed landscapes.

6.2.1 Prototype 1 - CDN/NPM

The first prototype developed was the CDN/NPM landscape. Totaling 8 micro frontends embedded as iFrames into a Luigi-Core-React application. Figure 6.1 visualizes this environment.

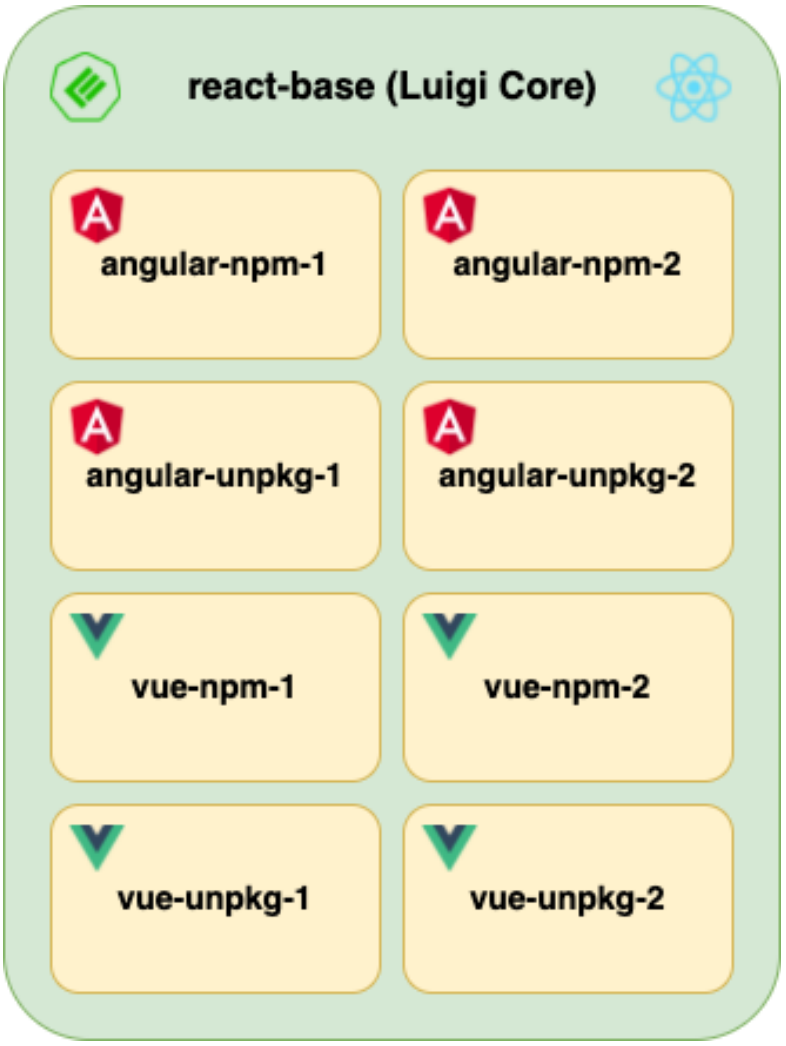

FIGURE 6.1: Overview of the Luigi prototype using Unpkg.com

This prototype represents a Luigi micro frontend landscape, where each Luigi node is a single-page application (SPA). The 4 NPM apps, embedded in the landscape, were implemented for comparison. They are meant to show, how the the CDN technology performs, in direct comparison to a bundled application. In their core, each application contains the same UI-elements. Thus, it is made possible to compare them, in terms of loading times and bytes loaded. The goals to show with this environment were:

- How redundancies occur in a micro frontend landscape
- That the browser can´t distinguish between bundled resources, contrary to resources loaded from the CDN
- If the UI framework, has an effect on the performance of the site

This landscape, fulfills 4 out of the 5 requirements, missing the aspect of different versions. The reason behind this is that, even though it is possible to request different

versions of a resource from the CDN, the CDN itself does not offer any means to resolve multiple versions inside the landscape. That means that the library is imported 2 times in different versions. That just increases the redundancies inside the landscape. Due to this predictability, this aspect was not considered for that landscape.

6.2.2 Prototype 2 - WMF/Web Components

The second prototype developed was the WMF and Web Components landscape. It contains 36 micro frontends in total, which are split over 6 Luigi Nodes. Each Luigi Node, is therefore a compound of micro frontends. Figure 6.2 provides an overview of the Node landscape.

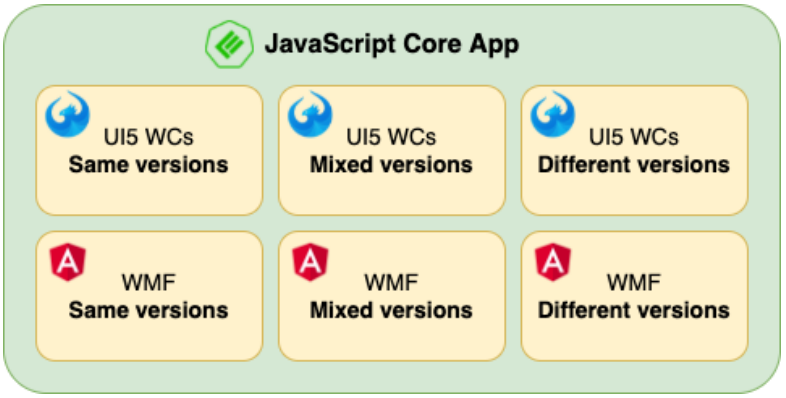

FIGURE 6.2: Overview of the Luigi Nodes in the WMF/Web Components environment

The 6 Nodes displayed in figure 6.2 contain 6 micro frontends each. Figure 6.3 provides an overview on how the micro frontends are arranged inside a Node to a compound.

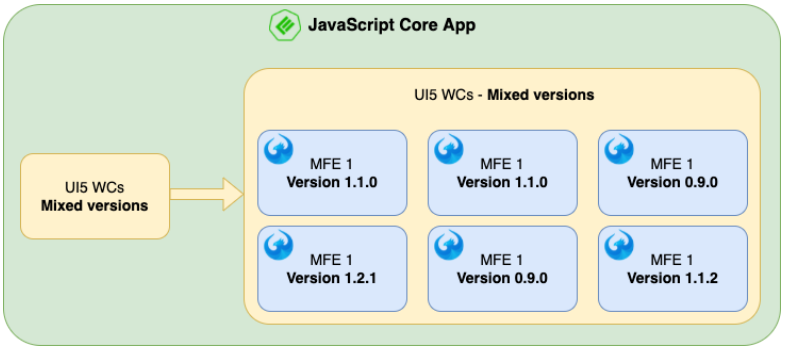

FIGURE 6.3: Example overview of a single Node inside the WMF/Web Components environment

Even though these two technologies are embedded in the same Luigi dashboard, they are referred to as separate landscapes. This is due to used technologies used for implementation. Nonetheless, this prototype fulfills 5 out of the 5 defined requirements for representation and therefore can be considered to be representative.

The following section will shortly explain, how the micro frontends are embedded in the Luigi Nodes and showcase their differences.

6.2.2.1 Embedding a micro frontend into a compound in Luigi

The micro frontends in the 3 Web Component Nodes are embedded as part of compounds. The according configuration is made in the `luigi.config.js`.[42] [43] Contrary to WMF, this configuration allows to embed the referred components in so called `compoundItemContainer`. The arrangement and number of columns inside the Node are defined in the `luigi.config.js`. Listing 6.1 shows the exact configuration for that.

```
Luigi.setConfig({
  navigation: {
    nodes: [{
      label: 'Home',
      pathSegment: 'home',
      icon: 'home',
      hideFromNav: true,
      globalNav: true,
      hideSideNav: true,
      defaultChildNode: 'mixedVersions',
      children: [
        // Mixed Versions
        {
          label: 'Luigi Compound mixed versions',
          pathSegment: 'mixedVersions',
          icon: 'activate',
          compound: {
            renderer: {
              use: {
                extends: 'grid',
                createCompoundItemContainer: (itemConfig,
                                              containerConfig,
                                              superRenderer) =>
                {
                  const itemContainer = superRenderer
                  .createCompoundItemContainer(itemConfig);
                  if (!itemConfig.noPadding) {
                    itemContainer.style = itemContainer
                    .getAttribute('style') +
                    ' ; padding: 20px ; overflow: hidden';
                  }
                  return itemContainer;
```

```
33                  }
34                },
35                config: {
36                  columns: '1fr 1fr 1fr',
37                  layouts: [{
38                    minWidth: 0,
39                    maxWidth: 1024,
40                    columns: '3fr',
41                    gap: 0
42                  }]
43                }
44              },
45              children: [
46              // MFEs for this compound go here
47              {
48                webcomponent: true,
49                viewUrl: 'URL to the view to be embedded',
50                label: 'A label for the MFE',
51                layoutConfig: {
52                  column: "auto"
53                }
54              },
55              ...
56              ]
57            }
58          },
```

Listing 6.1: Configuration of Web Components Node for a compound view

Line 14 to 16 define the central Node navigation element for the Luigi dashboard. In line 17 it is defined that the content of this Node is to be a compound. The following line 18 defines the renderer for the content and the tools are the properties of the `use` property in line 19. Line 20 to 34 define the actual CSS attributes of the to be rendered container for the compound elements. The arrangement of the container , is managed in the `config` property in line 35. In this scenario, the containers can be arranged in 3 columns, each one taking 1 fraction of 1024 pixels. At line 45 the actual compound elements or micro frontends are defined as an array of JavaScript objects.

6.2.2.2 Embedding a micro frontend into a compound in WMF

Contrary to the Web Component Nodes, the WMF Nodes are actually embedded as iFrames into the Luigi dashboard. Therefore, the Luigi configuration for that landscape,

is similar to the CDN/NPM environements. The listing 6.2, shows an example on how it was done.

```
Luigi.setConfig({
  navigation: {
    nodes: [{
      label: 'Home',
      pathSegment: 'home',
      icon: 'home',
      hideFromNav: true,
      globalNav: true,
      hideSideNav: true,
      defaultChildNode: 'sameVersions',
      children: [
              {
        pathSegment: 'wmfHostSame',
        label: 'WMF Compound Same',
        icon: 'technical-object',
        viewUrl: 'https://angular-wmf-same-shell.surge.sh/',
        loadingIndicator: {
          enabled: false
        }
      },
    ...
```

Listing 6.2: Example configuration to embed a WMF micro frontend compound as a Node in Luigi

As it can be seen, the integration is done via an URL. The same view displayed in the dashboard, can be found under the referred URL. This means, to create a compound view in WMF, the creation of the necessary container, the arrangement and the actual placements of the micro frontends, is managed by the host application of the WMF environment and not Luigi. The code for these development, is shown and explained in section 6.5 of this chapter.

6.3 CDN - Unpkg.com

Chapter 3, introduced the concept of the CDN. A remote server is provides the necessary platform resources via an API, thus avoiding the necessity of bundling those resources. To evaluate the impact of this technology on a micro frontend landscape, the first prototype was developed using the public cloud based CDN Unpkg.com. It is an open-source project, built and maintained by Michael Jackson. It runs on the Cloudflare platform,

and auto-scalable servers are provided by Fly.io, which are located in 17 cities around the world.[44]

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 usage of the Unpkg.com-CDN 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 6.3.

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

Listing 6.3: 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. The first part of the URL refers to the protocol and the host. The first path parameter is the npm dependency name. In the case of `??` 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 6.3, the `type="module"` attribute has to be added, 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.[45]

6.4 Web Components - UI5 Web Components

The Web Components standard was partially used in the development of the second prototype. For development itself no UI application framework was used, which means all three compounds were developed using plain JavaScript. 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.[46] Consisting of the picked elements, and under the usage of the scoping feature of the component library, the 3 compounds where created. The scoping feature in particular, made it possible to scope the tags of the UI5 Web Components, this way a multi version landscape was simulated.[33]

```
import { LuigiElement, html }
from "@luigi-project/client/luigi-element.js";
import { setCustomElementsScopingSuffix }
from "@ui5/webcomponents-base/dist/CustomElementsScope.js";
setCustomElementsScopingSuffix("placeholder");
import "@ui5/webcomponents/dist/Dialog.js";
import "@ui5/webcomponents/dist/Button.js";
[...]

export default class extends LuigiElement {
  constructor() {
    super();
    [...]
    render() {
      return html`
      <div>
      <div class="header">
      <h2>Products table - Version: placeholder</h2>
      </div>

      <ui5-table-placeholder id="ui5-table"
      ui5-table sticky-column-header>
      <!-- Columns -->
      <ui5-table-column-placeholder
      slot="columns"
      style="width: 4rem">
      <span
      style="line-height: 1.4rem">
      Product
      </span>
      </ui5-table-column-placeholder>

      [...]

      </ui5-table-placeholder>
      </div>`;
    }
  }
```

Listing 6.4: Scoping feature used in the prototype

Listing 6.4 shows the exact usage of the scoping feature. The imported *setCustomElementsScopingSuffix* function enables to define a custom suffix to all UI5 Web Component elements. 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 similar looking micro frontends, using 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, without the scoping feature, the *Bar* element in version *1.0.1* and *1.1.0*, are usually registered under the same tag name. By using the scoping feature, this conflict can be resolved. Through defining a version suffix for the *Bar* element, the elements are registered under distinguishable tags in the browser. 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 6.5.

```
1  import 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 6.5: Content of the `rollup.config.js`

The configuration of the bundler was split into several JavaScript files, to improve the readability of the file itself. The content of such a file can be seen in listing 6.6.

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

```
40      compact: true
41    },
42    plugins: [
43    replace({
44      'placeholder': '1-1-0',
45    }),
46    terser(),
47    resolve(),
48    json(),
49    url({
50      limit: 0,
51      include: [
52      /.*assets\/.*\.json/,
53      ],
54      emitFiles: true,
55      fileName: "[name].[hash][extname]",
56      publicPath: "\" + new URL(\".\", import.meta.url) + \"", //
      relative configuration for assets (TBD with UI5 Web Components
      team)
57    })
58    ]
59  },
60  [...]
61  ]
```

Listing 6.6: Configuration content for the `rollup.config.js`

The file, which is bundled for each configuration, in the shown listing 6.6, is always the same. This can be taken from the `input` property. The bundling result though, is named differently, which can be read from the `output` property. The first bundle result is called `tableViewMixedVersions1.js`, the second `tableViewMixedVersions2.js` etc. The `replace` method, in e.g. line 17, of the first configuration object replaces a string inside the input file. In the shown case, the string *placeholder* is first replaced with *0-9-0* and in line 43 with *1-1-0*. This leads to, that the elements inside those files are named and later registered under different tags in the browser. Therefore, elements registered by the first micro frontend are called for example `<ui5-bar-0-9-0>` and for the second then `<ui5-bar-1-1-0>`.

This results in, one file is bundled several time. Each bundle result, has a different name and replaces the placeholder string with a different string. This way, multiple micro frontends can be generated, which then again register the same elements under differently named tags. Therefore, those elements are treated as new elements.

6.5 Webpack Module Federation with Angular

The three implemented versions for the the WMF landscape are similar to one another, the only difference can be found are the dependencies and their configured sharing. Listing 6.7 shows the configuration for the *sameVersions* Node.

```
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 6.7: Content of `webpack.config.js` of the shell of the sameVersions WMF project

As it is shown, 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 not configured in this file. To add the dynamically loaded remotes, a service had to be developed which imports the remotes at runtime [47]. This service is shown in listing 6.8.

```
1  [...]
2  @Injectable({ providedIn: 'root' })
```

```
3  export class LookupService {
4    lookup(): Promise<PluginOptions[]> {
5      return Promise.resolve([
6      {
7        remoteEntry: 'https://angular-wmf-same-mfe1.surge.sh/
     remoteEntry.js',
8        remoteName: 'mfe1',
9        exposedModule: './Mfe1',
10
11       displayName: 'Mfe1',
12       componentName: 'Mfe1Component'
13     },
14     [...]
15     ] as PluginOptions[]);
16   }
17 }
```

LISTING 6.8: Content of `lookup.service.ts` for remote module loading in shell applications

This service serves the information of the remotely loaded modules to a proxy plugin, which is responsible for the actual rendering of the components. This proxy plugin is responsible for creating the container, as some sort of placeholder for the remotes. [47]

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

Line 8 of listing 6.9 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.[48] 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 `PluginOptions`, was defined in an interface, exporting a type definition. The code for it, is displayed in listing 6.10.

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

Listing 6.10: 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 6.10.

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 (aka remote) can be seen in listing 6.11.

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

As mentioned in chapter 5, to share dependencies every remote has participate. 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 `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` 6.8. 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 JavaScript object, which maps the path to the actual component. In the shown case, it is mapped to `./Mfe1`.[35]

In the section 6.2.2.2, it was shown how the developed WMF nodes are embedded in the Luigi dashboard. The shown snippets in this section have shown how a compound of micro frontends is created using WMF. In direct comparison with the Luigi compounds, it becomes obvious, that WMF does not really offer support for heterogenic landscapes. This is shown indirectly in the implemented prototype, but can be applied to any WMF landscape, too. Through the necessity of a separate service for rendering the micro frontends as part of a compound, it is shown that, WMF does not offer any functionality for this case. Additionally, the only reason this prototype was implemented without conflicts is, because each remote is actually an Angular project/component. If for instance, one project would be a VueJS based remote, the integration of it would be connected to more obstacles. This can be showcased, in lines 21 and 24 of the plugin proxy from listing 6.9, which responsible to load a remote component into the created container. The methods used in those lines, expect an Angular component. If now a VueJS remote would be loaded into, a conflict occurs. Furthermore, routing inside a compound view of WMF, requires further workarounds, even in a homogenic landscape. For example, in the case of the implemented prototype, the routers of the Angular micro frontends are not shared with the other members of the compound. Therefore, a router would have to be either loaded via an API or a dynamic import by the other participants or else routing between the micro frontends is impossible. Additionally, enabling the `webpack.config.js` and with it the Module Federation, is different for every framework. In case of Angular it was via the installation of the mentioned dependencies.

7. 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 8.

7.1 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.

7.1.1 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]

[1] surge.sh is a platform for static web publishing for frontend developers via the command line.

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.

7.1.2 Testing process

Since multiple landscapes were implemented a unified process was required to collect comparable data. The BPMN diagram 7.1 contains the process steps in detail.

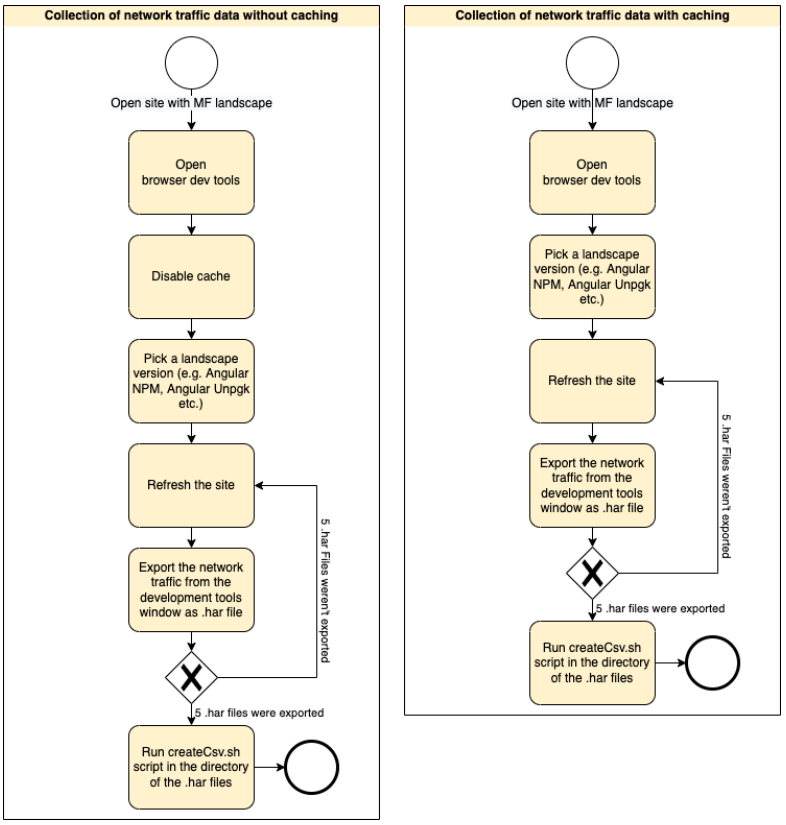

FIGURE 7.1: Data collection process for .har files

As it can be seen in 7.1 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 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

[2] This part of the environment was explained in chapter 3

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.

In summary it means:

- 12 landscapes were developed
- Each landscape was loaded 10 times, 5 times with cache enabled and 5 with cache disabled
- Each loading of a landscape is represented by a HTTP Archive file
- In summary, the testing processes were executed 120 times overall
- Generating 120 `.har` files
- Which then were aggregated and transformed into 12 `.csv` files
- Which then again were copied into one Excel file, where the data was analyzed
- In addition 12 Lighthouse reports where collected (one per landscape) and added to the excel

The reason why this testing procedure was chosen was, due to the low variety in the parameters when a landscape was loaded. First it was tested if the parameters would vary when a landscape is loaded up to 1000 times, but that case did not occur. Due the constant resource sizes and stable network, not many fluctuations could be observed. Only when changes on technological level were made (e.g. usage of cache, network outage) fluctuations started to appeared. But other than that, it was observed that the results were stable. Therefore it was decided that loading a landscape 10 times does suffice for the experiment.

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.

7.1.3 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.[19] 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.

7.2 CDN landscapes

7.2.1 Implementation with NPM

The table 7.1 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 7.1: 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

Table 7.1: continued from previous page

Metric	Angular NPM	Vue NPM
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 7.1 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 7.2: 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
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 7.2 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.

7.2.2 Implementation with Unpkg

Table 7.3 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 7.3: 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
Avg. Time in MS	97.23	194.01
Avg. Content Size	9768.62	8205.19

The data shown in table 7.3 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 7.4: 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

Table 7.4: continued from previous page

Metric	Angular Unpkg	Vue Unpkg
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 7.4 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.

7.3 Web Components/Compound and WMF landscapes

As the described in chapter 6, 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 7.5 shows the above mentioned KPIs of those environments.

TABLE 7.5: 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

Table 7.5: continued from previous page

Metric	WC/Compound	WMF
Avg. Content Size	213266.74	507168.76

As the data in table 7.5 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 7.6 showcases the same landscapes, but with caching enabled.

TABLE 7.6: 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
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.

7.4 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 7.7: 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

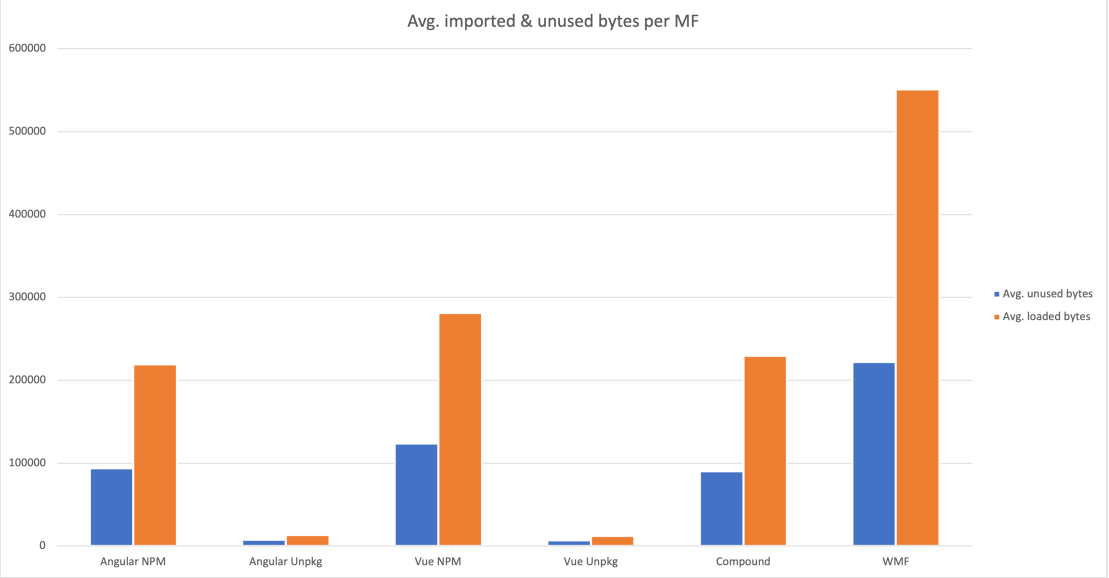

FIGURE 7.2: Unused and imported bytes per landscape

The shown data in table 7.7 and figure 7.2 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.

8. Conclusion

The following chapter will conclude and evaluate the results shown in chapter 7. 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.

8.1 Direct evaluation of the landscapes

8.1.1 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 7.2 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.

8.1.2 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.[49]

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.[49]

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.[49] On that regard Web Components offer easier means and ways for implementation due to their standardized aspects.

8.2 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 as long as it involves a homogenic landscape [1].

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 its 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.

[1] This was explained in section 6.5 of chapter 6

9. 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] 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.

[2] 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.

[3]

[4] 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.

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

[6] SAP SE. SAP History. https://www.sap.com/about/company/history/1972-1980.html. Accessed: 21.03.2022.

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

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

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

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

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

[12] MDN Web Docs. HTTP caching. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching, March 2022. Accessed: 21.03.2022.

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

[14] Sipat Triukose, Zhihua Wen, and Michael Rabinovich. Measuring a commercial content delivery network. New York, NY, USA, 2011. Association for Computing Machinery.

[15] 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.

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

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

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

[19] 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.

[20] imperva.com. Image Optimization. https://www.imperva.com/learn/performance/image-optimization/. Accessed: 21.03.2022.

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

[22] 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.

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

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

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

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

[27] Simon Rey. Conception and prototypical implementation of a web components based library in the context of reusability. In *Conception and Prototypical Implementation of a Web Components Based Library in the Context of Reusability*, page 8–10, 2021.

[28] 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.

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

[30] 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.

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

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

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

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

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

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

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

[39] 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.

[40] 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.

[41] 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.

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

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

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

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

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

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

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

[49] 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.

List of Figures

List of Tables

Glossary

API	**A**pplication **P**rogramming **I**nterface
WMF	**W**ebpack **M**odule **F**ederation
WC	**W**eb **C**omponent
CDN	**C**ontent **D**elivery **N**etwork
DOM	**D**ocument **O**bject **M**odell
PoP	**P**oint **o**f **P**resence
RTT	**R**ound **T**rip **T**ime

A. Results for all prototype landscapes

B. Lighthouse result table

C. Lighthouse result graphs