HO
GENT

Departement IT en Digitale Innovatie

Enabling the Distributed Development of Blazor-Based Web Applications Using a Microfrontend Architecture

Dante De Ruwe

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Benjamin Vertonghen
Co-promotor:
Dr. Florian Rappl

Instelling: smapiot GmbH

Academiejaar: 2020-2021

Derde examenperiode

Department IT and Digital Innovation

Enabling the Distributed Development of Blazor-Based Web Applications Using a Microfrontend Architecture

Dante De Ruwe

Thesis submitted in partial fulfilment of the requirements for the degree of professional bachelor in Applied Information Technology

Promotor:
Benjamin Vertonghen
Co-promotor:
Dr. Florian Rappl

Institution: smapiot GmbH

Academic year: 2020-2021

Third examination period

Preface

This bachelor's thesis was written as a part of the final steps of obtaining my professional bachelor's degree in *Applied Information Technology* at Ghent University of Applied Sciences and Arts (HOGENT). During the past years, I discovered my passion for software architecture and development, mainly focussed on .NET-related technologies.

My introduction into the wonderful world of microfrontends started when applying for an international internship in the summer of 2020. I came into contact with the company "smapiot" and one of their solution architects, Florian Rappl. He introduced the concept of the microfrontend architecture to me, mentored me through my internship entirely focussed around that architecture, and provided me with the inspiration for this thesis.
Microfrontends were right outside of my comfort zone. I was most familiar with backend development and architecture, and had almost no idea there was a need for distributed architectures in the frontend space to begin with. The international internship really pulled me deeper into the topic, and provided me with hands-on experience.

I want to sincerely thank Florian for his mentorship during the internship, for being the co-promotor of this thesis, and for providing me with useful guidance and sources.

Many thanks also to Benjamin Vertonghen for being the promotor of this thesis. He provided me with useful feedback and was available for my questions even during the vacation period.

Next, I want to take this opportunity to thank my family – with in particular my mother, father, sister, and stepfather – for mentally, practically and financially supporting me throughout my entire study career. Without them, I would not be where I am today.

Finally, a big "thank you" to my girlfriend for her continued love and support, particularly throughout the COVID-19 pandemic, the 3-month-long internship abroad, and the process of writing this thesis.

Samenvatting

Door de introductie van de WebAssembly *(WASM)* standaard is een hele nieuwe waaier aan webapplicaties mogelijk. Binnen het .NET ecosysteem is Blazor het meest gebruikte *framework* om WebAssembly applicaties te genereren. Dit framework maakt het mogelijk om interactieve *user interfaces* te bouwen, en hierbij gebruik te maken van C# in de plaats van JavaScript. Een uitdaging die bij deze manier van werken komt kijken, is dat er geen rechtstreekse manier is om een gedistribueerde ontwikkeling mogelijk te maken. Er kan gebruik gemaakt worden van onafhankelijke *component libraries*, maar dan moet de uiteindelijke web applicatie kennis hebben van deze *libraries* bij de integratie. Een *microfrontend* architectuur kan deze relatie potentieel omkeren.

Deze bachelorproef onderzoekt wat noodzakelijk is om Blazor applicaties gedistribueerd en op grote schaal te ontwikkelen. Hoewel de voordelen van de *microfrontend* architectuur grotendeels overlappen met die van andere gedistribueerde architecturen zoals de *microservice* architectuur, heeft een theoretisch onderzoek de uitdagingen die eigen zijn aan de *microfrontend* architectuur kunnen blootleggen. Een proof-of-concept applicatie werd daarna gecreëerd rond een specifieke *business case*, waarbij onder andere gefocust werd op zgn. *progressive enhancement*. Om dit te bereiken werd een universele compositiestrategie gehanteerd. Een herbruikbare *framework library* werd ook ontwikkeld, die voor een Blazor applicatie enkele nuttige componenten kan bieden om dynamisch zgn. *fragments* weer te geven en *client-side routing* mogelijk te maken. Ook werd hierbij gedacht aan de mogelijkheid om te *debuggen*.

Aangezien het gebruik van Blazor en de *microfrontend* architectuur momenteel in de lift zit, kan deze bachelorproef een startpunt zijn voor verder onderzoek, of een nuttige bron van informatie voor .NET ontwikkelbedrijven en -teams die op grote schaal *full-stack* webapplicaties ontwikkelen.

Abstract

With the WebAssembly *(WASM)* standard a new set of web applications have been made possible. Within the .NET ecosystem, the most popular solution for generating WebAssembly applications is called Blazor. This framework allows building interactive web UIs using C# instead of JavaScript. One challenge in this approach is that there is no direct way of enabling distributed development. While component libraries can be created independently, knowledge in the main application would be required for integration. Using a microfrontend architecture this relationship could be inversed.

This thesis investigates what is needed to empower the distributed development of large-scale Blazor-based web applications. While the advantages of the microfrontend architecture pattern are closely related to those of other distributed architectures such as the microservice architecture, a theoretical study uncovered that microfrontends have their own specific set of challenges that need to be overcome. A proof-of-concept solution focussed around a realistic business case was constructed, focussing on progressive enhancement. To achieve this, a universal composition strategy was used. A reusable framework library was created that can provide any Blazor application with components to achieve dynamic fragment rendering and client-side routing, keeping also the debugging experience in mind.

As the adoption of the Blazor framework and the microfrontend architecture pattern will mature, this thesis could be a valuable starting point for further research, and a valuable resource for .NET-focussed development teams creating large-scale full-stack web applications.

Contents

List of Figures

Glossary

API
: An API or application programming interface describes a set of commands and protocols for the communication between software systems without having to know their exact implementation. 14, 27, 28, 39, 48

API gateway
: An API gateway accepts API requests from a client, and directs them to the appropriate services. Typically it handles a request by invoking multiple microservices and aggregating the results (NGINX, 2021). 29

application shell
: Also called app shell. It serves as a parent application for the integration of microfrontends (Geers, 2020)(Rappl, 2021). 36, 38, 39, 44, 47, 48, 50

backend
: The backend of a software application is the part of a software system that is not directly accessed by the user, typically responsible for executing business logic and manipulating data. 3, 19, 27–29, 32, 33, 37

BFF
: Backend For Frontend. 29, 30

CDN
: A Content Delivery Network, is a distributed network of servers that can efficiently deliver web content to users (Microsoft, 2018). 36, 48

CLI
: Command Line Interface. 47

DI
: Dependency injection. 47

DLL
Dynamic-link library. 48

DOM
Document Object Model, a programming interface for web documents. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page (Mozilla, 2021a). 35, 36, 39

endpoint
Describes the point of entry in a communication channel. Relating to APIs, an endpoint is the location or address where a request can be made to. 27, 28

ESI
Edge-Side Includes. 36

frontend
The frontend of a software application is the design, architecture and programming that makes the user-facing application function. 3, 19, 20, 26–28, 32, 33, 37, 45

FTS
Follow-the-sun (also called 24-hour development or round-the-clock development) is a form of GDD in which multiple teams are spread across timezones to ensure one team is always operational during normal business hours. 24

GDD
Geographically Dispersed Development. *See* GDD. 23

GDD
Geographically Distributed Development. The practice of managing software development projects beyond the traditional bounds of a single building or office structure where the development staff is singularly located. In a GDD model, the development staffing may be distributed across town, across a state or provincial border, or overseas (Yuhong, 2008). 23, 24

GSD
Global Software Development. *See* GDD. 23

GUI
Graphical user interface. 25, 26

HTTP
The Hypertext transfer protocol is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, request/response protocol (Fielding et al., 1999). 28, 29, 48

iframe
Inline frame. 35, 37

IL
Intermediate language, also known as MSIL, is a product of compiling high-level .NET languages into a binary instruction format (Microsoft, 2016). 39

JS interop
Javascript interoperability describes the invocation of a JavaScript code from another language. 39

load-balancer
A load enables the optimization of computing resources, reduces latency and increases output and the overall performance of a computing infrastructure, by distributing and managing the load across several devices (Techopedia, 2012). 28

microfrontend
See microfrontend architecture. 3, 5, 7, 19, 20, 33–38, 40, 43–50

microfrontend architecture
The microfrontend architecture is an architecture style that splits up an application into distributed modules that are focussed around a specific business capability. Most of the time these individual modules are managed by autonomous cross-functional teams (Geers, 2020)(Rappl, 2021). 3, 7, 15, 19–21, 23, 32–35, 37, 38, 41, 43, 49

microservice
A microservice is a cohesive, independent process interacting via messages (Dragoni et al., 2017). Every microservice is loosely coupled, independently deployable and organized around a business capability. *See also* microservice architecture. 5, 15, 20, 29–32, 35, 37

microservice architecture
A microservice architecture is a distributed application where all its modules are microservices. (Dragoni et al., 2017).*See also* microservice. 7, 15, 28–33, 37, 38

monolith
A software monolith or monolithic application describes an application that is built in a single unit, and that produces a single logical executable. This means any changes that are made to a part of the application require building and deploying a new version of the application (Fowler & Lewis, 2014) 15, 19, 20, 26–28, 32

monolithic
Built as a monolith. *See* monolith. 19, 20, 23, 28, 32, 41

MVC
Model-view-controller. A design pattern, commonly used for user interfaces with the goal of seperating the user interface from the underlying data that represents it (Leff & Rayfield, 2001). 25, 26

offshoring
The tranfer of a business function to another country, usually to take advantage of lower labor costs, tax rates, or for legal reasons. 24

outsourcing
The transfer of a business function to a third-party service provider. 23

PDB
Program database. 40, 48

PoC
Proof of concept. 43–46, 48

progressive enhancement
A software design philosophy that emphasises the delivery of simple content and basic funtionality to as many users as possible, while providing the best user experience only to the users of capable systems and browsers. 5, 7, 20, 35, 37, 43

REST
Representational state transfer is an architectural style for distributed hypermedia systems. When a RESTful API is called, a transfer between the server and the client will occur that represents the state of the requested resource (Avraham, 2017). first 16, 29

RESTful
See REST. 27, 48

RPC
A Remote procedure call, also known as a subroutine call or a function call, is a communication mechanism for client-server applications. 29

SEO
Search engine optimization. 35–37, 44

SOA
Service-oriented architecture defines a way to make software components reusable and interoperable via service interfaces. Each service in an SOA embodies the code and data required to execute a business function (Education, 2021). 28

SOAP
Simple object access protocol, a method of transferring messages formatted in XML over the Internet. 28

SPA
Single-page application. 36

SSI
Server-Side Includes. 35, 36

SSO
Single sign-on. 29

transpiling
Taking source code written in one language and transforming into another language that has a similar level of abstraction (high-level to high-level or low-level to low-level) (Fenton, 2012). 38

UI
User interface. 7, 19, 38, 49

URL
Uniform Resource Locator. The address of a specific webpage or file on the internet. 35

W3C
The World Wide Web Consortium is an international community where Member organizations, a full-time staff, and the public work together to develop Web standards (W3.org, 2021). 38

WebAssembly
WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications (Webassembly.org, 2021). 5, 7, 19–21, 23, 38–41, 47, 49

XML

Extensible markup language is a markup language for representing structured information that is both human and machine-readable. 28, 36

1. Introduction

Historically, nearly all applications were developed using a *monolithic* architecture: a single-tier architecture, where the user interface (UI), business logic and data storage are all managed in a single all-in-one solution. Nowadays, developer teams have mostly adopted split-stack development, where the UI is handled in the so-called "frontend" and the business and data logic is dealt with by a "backend" system. This reduced coupling enables specialized teams to develop each aspect individually, independently and therefore simultaneously (Dunkley, 2016). For the same reasons, *microservices* started making an appearance when developers realized that having a single backend service could also be considered a monolithic approach (Fowler & Lewis, 2014).

In very recent years, the term "*microfrontends*" (or "*micro frontends*"), was introduced to describe the split of the frontend monolith into independently deployable and maintainable pieces. The resulting web applications are often described as "modular distributed web applications", and allow autonomous cross-functional teams to work on them.

This "vertical" split is especially beneficial for large-scale projects, where the division of developers in autonomous teams is quite common. However, enabling the distributed development of these microfrontends is not a trivial undertaking, and naturally has its challenges.

This thesis aims to be an application of the microfrontend architecture, to enable distributed development of large-scale web applications. More specifically, this thesis will focus on the Blazor[1] WebAssembly[2] framework.

[1] https://blazor.net
[2] https://webassembly.org

1.1 Problem Statement

The introduction of the microservice and microfrontend architecture patterns gives rise to technical and organizational challenges that were previously less commonly encountered in the world of monolithic development models. However, there are significant benefits too if development teams are willing to adapt to these methodologies.

The research in this thesis can indicate whether development companies and teams can benefit from a move towards one of these distributed architectures, or if the benefits don't outweigh the difficulty of the changes.

Furthermore, while *technology independence*, for example, can be categorized as a benefit of said architecture patterns, this does not automatically ensure that every technology is well suited.

This thesis focusses on the specific characteristics of the Blazor WebAssembly frontend framework in relation to microfrontends. Because of the limited maturity of Blazor and the application of the microfrontend architecture pattern within this tech stack, this thesis could be a valuable resource to companies and development teams that have a focus on Blazor development or .NET technologies in general.

1.2 Research question

To be able to achieve the research objectives, the following research questions were formulated:

RQ_1 What is needed to be able to independently develop and deploy microfrontends using Blazor WebAssembly?
RQ_2 How to render Blazor-based microfrontends with the proper isolation, performance and with progressive enhancement in mind?
RQ_3 What are the challenges that need to be overcome, and how would one do so?
RQ_4 How can development teams benefit from the transformation of their Blazor monolith into a microfrontend solution?

1.3 Research objective

- A comparative study providing an overview of the microfrontend architecture and the key differences with a monolithic architecture.
- A descriptive study outlining implementation patterns, challenges and best practices of the microfrontend architecture pattern in a Blazor WebAssembly project.
- A descriptive study outlining the benefits and drawbacks of using the microfrontend architecture with the goal of enabling distributed development in a company context.

With the gained insight of the literature and the theoretical study, a proof-of-concept solution around a realistic business case will be created to investigate the feasibility and

demonstrate the practical application of the microfrontend architecture pattern in a Blazor WebAssembly project.

1.4 Structure of this bachelor thesis

De rest of this bachelor thesis is outlined as follows:

In Chapter 2 an overview of the state of the art within the research domain is provided. This overview is based on a literature study.

In Chapter 3 the methodology is clarified and the relevant research techniques are discussed to be able to formulate an answer to the research questions.

In Chapter 4 a proof-of-concept solution will be created to be able to answer to the research objective of investigating the feasibility and demonstrating the practical application of the gained insights from the theoretical studies.

Finally, in Chapter 5, the conclusions about the findings are described and interpreted. Additionally, the significance of the results are outlined, which could provide incentive for further research.

2. State of the art

In this chapter, the research domain will be explored in its current state. This will allow for further insight into what distributed development is and why it is beneficial. The evolution of software architectures from monolithic to distributed will also be laid out, which introduces the concept of the microfrontend architecture. To conclude, a deep-dive into Blazor WebAssembly will be presented.

2.1 Distributed development

In a technology landscape that has long been shifting to meet the current trends of economic globalization, software development has evolved from being mostly concentrated at a single location, to being geographically distributed around the globe.

According to Yuhong (2008), the terms Geographically Distributed Development (GDD), Geographically Dispersed Development (GDD) and Global Software Development (GSD) are mostly used to refer to the same distributed development model. In the remainder of this thesis, the term Geographically Distributed Development (GDD) will be used mainly when emphasis is required on the geographical aspect of the distributed development model.

The application of GDD can be done in many distinct ways. One way is in the form of **outsourcing** agreements, often with countries where employment costs are more economically beneficial.

Another way, is the separation of a company into different local **divisions** or departments in different cities. This envelops both large multinational companies that operate around the globe, or more nationally-focused companies that have different branches in different

cities. (Kiel, 2003) This also includes the practice of **offshoring**, which tends to have the same motivations as outsourcing, but keeps control in the hands of the business, and does not involve a third party (Oshri et al., 2015).

2.1.1 Benefits

Companies often have a plethora of different business reasons to distribute the development, maintenance and management of software.

- Financial benefit (labor costs, taxation, ...)
- Market insight: local teams have more insight into local trends
- Talent availability: a larger pool of skilled developers is available, potentially even with different specializations in different locations. (Conchúir et al., 2009)
- Faster time-to-market: because development can be distributed across multiple timezones, a follow-the-sun (FTS) workflow could potentially increase development speed drastically[1] (Carmel et al., 2010).

2.1.2 Challenges

While the aforementioned benefits of GDD appear very useful in theory, on the practical side, there are of course some challenges to this approach. For example, according to Šmite et al. (2010), implementing an **agile methodology** within a distributed software development model is not straightforward; and the characteristics of agile and distributed development could be seen as polar opposites.

But the idiomatic *"elephant in the room"* is **communication**. In projects where teams are located together, communication can be rather informal, which helps team members more rapidly gain project and technical knowledge, as well as knowledge of the more human aspects of their coworkers, such as working style and expertise. According to Sengupta et al. (2006), frequency of communication has an inverse relationship to physical separation of team-members, and in multi-site environments, the decrease in communication frequency is so sharp, that informal communication is nearly nonexistent. Pairing this with **cultural** and timezone differences, makes all communication very difficult when practicing GDD.

2.1.3 The role of the architecture in enabling distributed development

According to Yuhong (2008), in a GDD environment, establishing and maintaining a common software and/or solution architecture that can support a distributed development model, is key for the success and sustainability of the software project. Among other architectures, she describes a module-based project architecture, where self-contained software components are developed independently. This way, teams could develop these modules simultaneously, without a large interdependence on other modules. This, however,

[1] According to Conchúir et al. (2009), FTS workflow is however practically almost inachievable, and in practice, many companies even make sure the time zones overlap as much as possible, to reach better inter-team communication.

requires strong decoupling of the software modules.

2.2 The evolution from monolithic to distributed architectures

The development of applications for the web has seen some dramatic shifts over the years. Apart from new technologies, protocols, and standards, the way web applications are structured has undergone some evolutions as well. "Software architecture" not only outlines the pure structure of the application but also defines the responsibility of all the pieces of the application, and how these pieces ought to interact with each other (Fedorov et al., 1998).

2.2.1 Monolithic architecture

Historically, one of the earliest architectures for implementing web applications was the **client-server model**. The service provider or *server* can share its resources with service users called *clients*. According to Reese and Oram (2000), at the time, most web applications were simple two-tier client-server applications. The web browser on the client-side retrieves data and files from the data store at the webserver side, without much data interpretation or manipulation. The upcoming increase of the computing power of hardware, made it possible to execute some data processing on the client-side, using for example technologies like Java.

Although two-tier client-server architectures were quick to set up and had robust tooling, a pretty significant downside got introduced: these so-called *fat clients* were now not only concerned with the task of presenting data to the user, but are also bloated with business logic and data processing. Conversely, any change in business rules would also require every client to adapt to this change. (Gallaugher & Ramanathan, 1996).

A **three-tier** architecture was conceptualized in an attempt to overcome the downsides of the two-tier approach. The idea sounds not very groundbreaking: just introduce the handling of business logic on the server-side, rather than on the client. This results in the following tiers (Aarsten et al., 1996):

- The client tier (also known as the presentation tier), which contains the graphical user interface (GUI)
- The application tier (also known as the business tier or logic tier), i.e. the application servers that contain objects representing business entities and domain logic.
- The data(base) tier that handles the storage of domain objects and data.

These architectures are compared in Figure 2.1.

With the introduction of this physical separation, the next challenge was the structure of the software's code.

Over an extensive period, applications built on top of the client-server model extensively used the **model-view-controller (MVC) pattern** (Pavlenko et al., 2020). Like the name suggests, this pattern describes 3 parts, that are used as conceptual and architectural

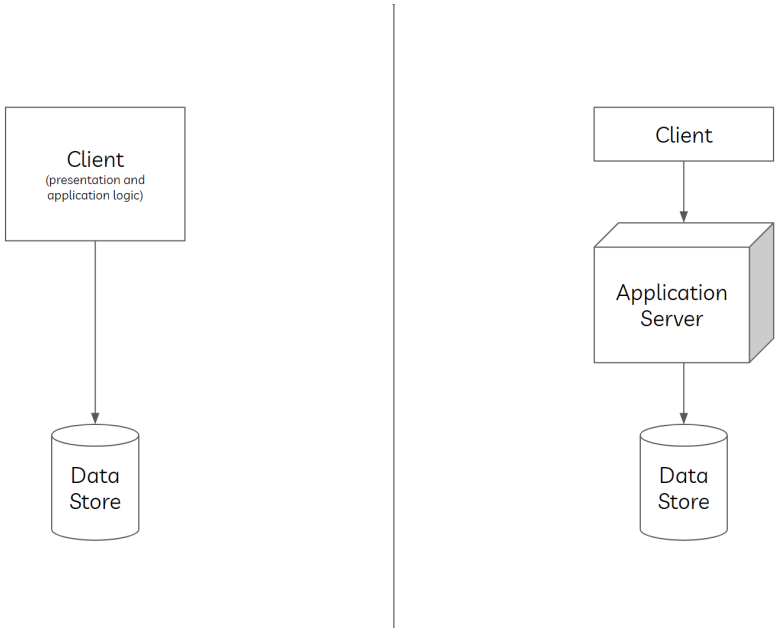

Figure 2.1: A diagram showing the difference between a two-tier *(left)* and three-tier *(right)* architecture. The two-tier architecture communicates to the data store without an intermediary application server, unlike the three-tier architecture.

separations in the software. The business logic of the application is encapsulated in the **model**, the presentation logic is the responsibility of the **view**, and the **controllers** handle the user actions in the views, and connect these actions to the appropriate models and updated views.

However, according to Leff and Rayfield (2001), implementation of the MVC pattern for web applications in a client-server environment, brings up the question of partitioning between servers and clients. Instinctively it is clear that the views belong to the client and the models belong to the server, but for the controllers, this separation is not so clear. Assigning the controllers to the client-side would result in a *fat client* again, while assigning them to the server-side (the *thin client* approach) would often mean too many round-trips to the server must be performed on every request. In practice, the workaround for this partitioning is a *dual MVC* approach, which partitions the controllers between the client and the server.

While the MVC architecture pattern can give developers a cleaner seperation of concerns in their code, it still produces a **tightly coupled** solution. Any change would still mean the entire code would have to be rebuilt and redeployed (Fowler & Lewis, 2014). This significantly reduces development speed, as well as agility. Monoliths can also be considered a "single point of failure": whenever one part of the software is malfunctioning, the whole system is crippled.

2.2.2 The split-stack development model

In more recent times, developer teams started to adopt **split-stack development**, where the GUI is handled in the so-called "frontend" and the business and data logic are dealt

with by a "backend" system. The communication between these two parts can be carried out in multiple ways, but one of the most common ways is via a (RESTful) application programming interface (API).

Decoupling the frontend presentation logic from the backend business logic introduces some major benefits: (Dunkley, 2016)

- **Multiple clients**
  As it is not at all concerned with presentation logic, the same backend service can now serve data to multiple different clients (e.g. web, desktop, mobile,...). To support a new type of client, only presentation logic has to be implemented.
- **Specialized teams**
  The split of frontend and backend allows developers to specialize themselves in these fields, yielding a deeper knowledge of the selected fields.
- **Independent technology stacks**
  Tying in with the previous point: as teams specialize, they desire and/or require more specialized technologies. The decoupling of front- and backend removes any restraints on technology selection between the two areas. For example, frontend developers can use client-side technologies and frameworks such as Typescript and React, while backend developers can leverage server-side technologies such as .NET or Java. This also makes each individual solution more future-proof.
- **Simultaneous development**
  As a result of all outlined benefits above, and the basic concept of decoupling, teams can work independently, autonomously and therefore simultaneously. For example, a change to the visual styling of a website, will not require any of the server-side code to change, and will also not trigger a rebuild.
- **Independent deployment and scaling**
  While a monolith has to be deployed as one big program, teams can now deploy the frontend and backend solutions according to their needs. Often a static hosting solution is enough to publish a frontend application, while a backend service might need some serious infrastructure to remain operational.

Of course, these benefits also come at the cost of some drawbacks.

Firstly, the fact that specialized teams can work on either the frontend and backend separately and in their own technology stack is a double-edged sword. While the benefits outlined above stay true; this also means that developers are less flexible to switch teams in times of developer shortage or approaching deadlines.

Secondly, the contract between the frontend and backend teams is a well documented API with clearly defined endpoints. For RESTful web API **documentation**, the OpenAPI specification[2], which is also known as *Swagger* documentation, provides a standardized way of defining this contract (Koren & Klamma, 2018). Of course, time and effort needs to be spent creating robust documentation.

Thirdly, breaking changes have to be avoided at all times. Frontend code might depend

[2] https://www.openapis.org/

on backend endpoints that will get deleted in an upcoming update. This is not acceptable. **Versioning**[3] provides a way to keep supporting clients that rely on a version of the API before the breaking change was made.

Lastly, more relevant in the context of this thesis: while dividing the frontend from the backend does provide benefits; in large, rapidly scaling applications, it simply isn't enough. After a frontend-backend split, one does still end up with a both a **monolithic backend** and a **monolithic frontend**.

Shifting the frame of reference to the server-side: the backend monolith still possesses the same downsides the overall monolith had introduced: one change in the logic and the entire application has to be rebuilt and redeployed. Also, the issue of scaling still persists: the server can be updated with more powerful hardware (vertical scaling) or the *entire* server-side application can be replicated on different servers, with a load-balancer deciding how to distribute incoming requests (horizontal scaling). Keyword here is *"entire"*, as no specific part of the server-side application can be scaled up independently.

2.2.3 Microservices

Due to an increased business interest in software services, the focus for choosing a software architectural style and paradigm shifted towards reusability and robustness. According to Dragoni et al. (2018), This was characterized by the shift to more modular, loosely coupled applications. The benefits of which were outlined in the previous section. The service-oriented architecture (SOA) pattern emerged as a step in this direction. Initially, the goal was to devise software services as a way to interface with larger software systems (which were often monolithic) via messages, using common messaging protocols (e.g. HTTP, SOAP or XML).

Taking this approach further, by not necessarily integrating with existing monolithic applications, but using the service-oriented approach to design, develop and deploy self-contained and autonomous software services, is what the microservice architecture is about.[4]

A visual overview of the microservice architecture is shown in Figure 2.2.

Principles of the microservice architecture

The microservice architecture is built on a few basic principles (Dragoni et al., 2018) (Dragoni et al., 2017) (Fowler & Lewis, 2014) (Gysels, 2020) (Newman, 2015):

- **Small codebases managed by small teams**
  Codebases are generally smaller in size, at least small enough to be managed by a small team (this typically means less than 10 people).

[3] https://restfulapi.net/versioning/

[4] While conceptually derived from SOA, labeling the microservice architecture as an implementation of SOA is highly debated. Read more on https://martinfowler.com/articles/microservices#MicroservicesAndSoa.

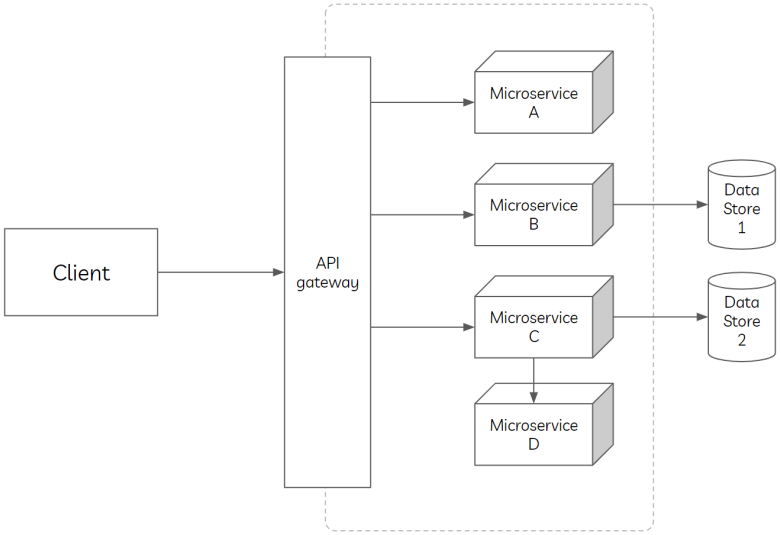

Figure 2.2: A diagram showing an example of a microservice architecture with services A through D.

- **Focused on a specific business need**
  Every microservice should be centered around a specific business need or domain, so that the responsibility of every microservice is clearly defined[5], and there is an alignment between the business capabilities and the software architecture.
- **Modular, decoupled and independent**
  According to Evans (2004), using a software model in a well defined *bounded context*, without worrying about the applicability of the model to other contexts, is key to keeping the context *pure* and avoiding confusion.
  Every microservice should be built, tested and deployed individually and in isolation.

The only form of **communication** between individual microservices is a uniform communication mechanism via network calls. In practice, the most commonly used mechanisms are the request-response mechanism using for example HTTP (with REST or remote procedure call (RPC)) or the event-based mechanism using an *event bus*[6]. The latter mechanism is generally encouraged as event-based collaboration is highly decoupled (Newman, 2015).

Communication with the "outside world" (i.e. with a client) is usually done using a facade over all the underlying microservices: usually called an *API gateway*. This service can aggregate content gathered from multiple backend calls, and serve it. Whenever this layer becomes too large, or becomes bloated with logic, a *Backend For Frontend (BFF)* pattern can be more useful: this is essentially equivalent to an *API gateway* per client. This is shown in Figure 2.3.

As for **authentication**, according to Newman (2015), a single sign-on (SSO) solution is often used, whereby the user can authenticate with an *identity provider*.

[5] This can be seen as the application of the Single Responsibility Principle to independent services
[6] e.g. Apache Kafka, RabbitMQ, Azure Service Bus, ...

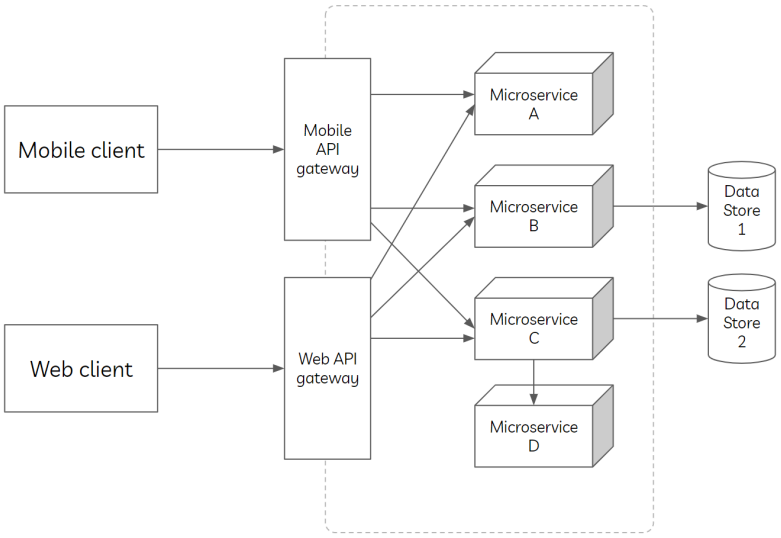

Figure 2.3: A diagram showing an example of the BFF pattern with a mobile and a web client.

Benefits of using a microservice architecture

Below some of the major benefits of the microservice architecture are outlined:

- **Technology independence**
  Explanation of this benefit can be directly borrowed from 2.2.2 ("*The split-stack development model*"). Choosing the right technologies for the problem is obviously very beneficial.
- **Scalibility and elasticity**
  Scaling a microservice architecture does not require reduplication of the entire software system. Every component can be scaled as needed with respect to their expected or measured load. Microservices are often deployed using *containers* using technologies such as Docker[7], which can be managed in *clusters* by other technologies such as Kubernetes[8]. This way, scaling can be done *elastically*: dynamically according to the load. (Dragoni et al., 2018)
- **Availibility and resilience to failure**
  While the methods used for scaling can improve performance and make it possible to cope with a high load on a software system, the same methods can also be applied to create redundancy and therefore resilience to failure. Also, because a defect in one microservice does not lead to the crashing or malfunctioning of the entire system. When errors do come forward, they are easier to locate in the narrow scope of the microservice. Additionally, when upgrading a microservice, instances of the old and new versions can run side by side, aiding in a smooth transition between them, and less or no downtime. (Dragoni et al., 2017)

[7] https://docker.com
[8] https://kubernetes.io

- **Better teams**
  The term "better" here is used in place of many different adjectives. Because of the limited scope a microservice has, teams gain a deeper understanding of the business domain, and feel a greater sense of ownership over the features they develop. Combining this with the smaller codebases and shorter release cycles, teams tend to get smaller and more productive (Newman, 2015).

Challenges and limitations of the microservice architecture

While lots of the benefits of the microservice architecture aid in enabling distributed development, the architecture cannot be regarded as a *silver bullet*. In what follows, some of the challenges of the microservice architecture are outlined, based in part on a literature review by Soldani et al. (2018). Depending on the perspective of the reader, some of these can also be considered to be drawbacks.

Partitioning the services tends to be a big challenge in the design phase of microservice development. It is often difficult to slice up the business capabilities into well-defined categories. Another challenge in the conceptual phase is establishing a clear strategy for **communication** mechanisms: microservice intercommunication should have clear contracts to ensure their compatibility.

During the development phase, because of the distributed nature of the microservice architecture, issues concerning data come up frequently. To ensure the necessary decoupling, each microservice should have a dedicated data store or database. If there is data duplication, this can raise issues with **data consistency**. Operating with distributed data stores also brings up the issue of **distributed transactions**: if one of the services fails to perform a database operation, what will happen to the principle of a transaction?

Another challenge in the development phase is designing and writing adequate **tests**. Because of their independent nature, unit tests are easier or in the worst case simply unaffected by the microservice architecture. Performance testing, integration testing and end-to-end testing, however, become more difficult because the system has to be tested from the outside. Often multiple services have to be spun up to be able to do a test properly, increasing the performance cost of the tests, and lowering their reliability.[9]

If the microservice-application is running, most of the challenges of the microservice architecture boil down to **increased operational complexity**: debugging, logging, monitoring, service coordination, etc.

It is also worth noting that the microservice architecture requires a certain **developer skill and knowledge**, that might not be readily available in a company or organization.

[9] See also https://blog.indrek.io/articles/challenges-of-testing-microservices/

2.3 Microfrontends

With the introduction of the microservice architecture pattern, backend systems can be split up into multiple services, each with their own responsibilities. As previously described, this can bring great benefit. However, even after a transition to microservices on the server-side, the client-side applications using these services are mostly still monolithic in nature.

A monolithic frontend does not have to be a problem. Monoliths are quick and easy to set up, and historically, most of the heavy lifting was done on the server-side anyway. However, the complexity of client-side applications has seen a drastic increase over the last few years. This can be attributed to many factors: increased hardware and web browser capabilities, a wide variety of client devices, massive market growth for digital services, and the web transitioning from a document platform to the largest application platform (Ball, 2019); just to name a few.

In these complex applications, the downsides of a monolithic architecture come back into view: every change requires the entire frontend application to be rebuilt and redeployed, codebases grow very large in size, etc... Even worse, because the client-side application has a functional dependency on the server-side application, a small change in one particular area of the backend logic could also trigger a change in the frontend, causing the entire frontend to yet again be rebuilt and redeployed (Rappl, 2019).

There is also the issue of **domain knowledge**: while the microservice architecture gives backend teams the possibility of focussing on one specific part of a business domain, teams that are developing the client-side code are still expected to know the entire scope of the application. Often this means a reliance on personal inter-team communication, which in larger organizations tends to be expensive (Geers, 2020).

While component-based paradigms introduced by libraries and frameworks (such as React[10], Angular[11] and Vue[12]) can alleviate some of the complexity of the current frontend systems, they still do not enable fully autonomous, decoupled, modular and/or distributed development of large web applications.

2.3.1 What are microfrontends?

In 2016 the ThoughtWorks Technology Radar (ThoughtWorks, 2020) coined the term "*Micro Frontends*" to describe the split of the frontend monolith into independently deployable and maintainable pieces. This new architecture pattern could therefore be regarded as an extension of the microservice architecture into the frontend space.

The characteristics of the microfrontend architecture pattern are therefore very closely related to those of the microservice architecture pattern, as described in section 6 ("*Principles of the microservice architecture*"). Every individual frontend module has a relatively

[10] https://reactjs.org
[11] https://angular.io
[12] https://vuejs.org

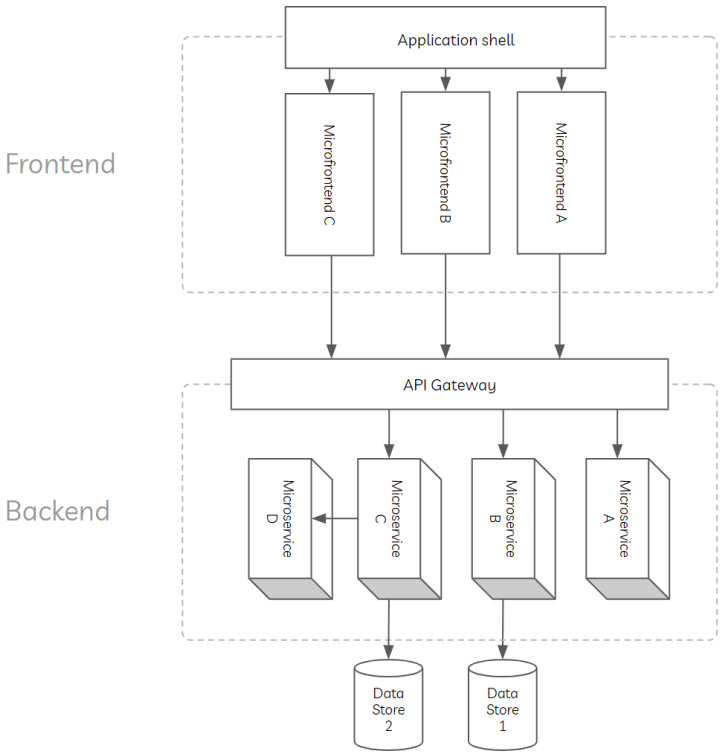

Figure 2.4: A diagram showing an example of a microfrontend architecture.

small codebase, is focussed around one specific domain or company mission, and should be modular, decoupled and independently developed; most optimally by autonomous teams.

A visual representation of the microfrontend architecture pattern is shown in Figure 2.4.

2.3.2 How do microfrontends enable distributed development?

A significant difference between microfrontends and other software architectures is the impact on **team structure** and organizational shift (Geers, 2020). So, while microfrontends have lots of technical aspects to consider, it is important to reflect on the organizational aspects first.

As discussed in 2.2.3 (“*Microservices*”), when adopting the microservice architecture, a shift has to be made to smaller independent teams around a specific business need. However, this only brings about changes in the backend teams, while the frontend-oriented development team will usually not follow suit. This keeps the overarching team structure “horizontal”: divided per layer or technology.

One of the benefits of horizontal teams is that this structure enables experts focussing on specific technologies to co-operate together as one team. This way, they can ensure a high technical standard within the boundaries of their respective areas of expertise.

In geographically distributed teams, the consequence could be that these technical teams operate from entirely different locations. According to Šmite et al. (2010), this can result

in compatibility issues between software layers and a less customer-focused development model. Moreover, disputes can arise between the different teams because no team has full responsibility for the delivery of any feature.

Feature teams

To mitigate the issues arising from geographically distributed horizontal teams; multidisciplinary or cross-functional "feature teams" can be introduced. These are grouped around a specific business case or customer need. This "vertical slicing" enables teams to be more independent and have end-to-end responsibility for the features they develop.

This feature team approach has multiple advantages: (Šmite et al., 2010)

- **Optimized feature development**
  Focussing on features instead of technical details aids in delivering the highest amount of business value.
- **Decreased need for "expensive" communication**
  Communication within a team is usually faster and more informal than communications between teams, especially if teams are geographically distributed. Since a feature can be developed by a single team, the need for expensive communication decreases.
- **Greater sense of developer involvement**
  According to (Larman & Vodde, 2008), developers who operate in a feature team feel a greater sense of ownership and accountability for the features they develop.

It is worth mentioning that these benefits combined can result in a faster cycle time and thus an increase in development speed (Geers, 2020).

On the other side, this approach also comes with caveats. One of them is the danger of compromising on the **conceptual integrity** of the software systems. More time will have to be spent up-front laying a solid foundation for conventions and standards. The effort to then maintain the consistency of the software system is usually where the software architect has a key role. In larger projects, a technical service team can be assembled to provide technical coherence across different distributed systems (Šmite et al., 2010). While this is something to consider, this also means an increased emphasis on better code quality right from the start, which could prove very beneficial in the long term (Larman & Vodde, 2008).

Another caveat is that, especially in large corporations, **changing the complicated organizational structure** might not be possible, or at the very least slow and difficult.

2.3.3 Common implementation patterns

As is the case with lots of architectural patterns, there are many ways of implementation possible. With the microfrontend architecture, this is also the case. Various options differ in complexity, goal, and mechanism.

Microfrontends often need to be composed to be able to reach a coherent application.

Every Microfrontend brings its own **pages**. Often, microfrontends also expose reusable components to be rendered in a variety of locations, to provide functionality. These are often called **fragments**.

Below some of the possible implementation and composition techniques are outlined.

The *web approach*

The most straightforward way of leveraging the microfrontend architecture pattern is by not using any integration technique at all. Instead, there can be opted for mechanisms that exist in the world of the web.

As described by Rappl (2021), the *web approach*'s main mechanism for microfrontend reference is by way of their Uniform Resource Locator (URL). **Hyperlinks** can be used to let the user navigate between different microfrontends. This, however, compromises on user experience and usability. With this approach, for example, elements from multiple microfrontends cannot be rendered on the same page.

To enable visual composition of microfrontends in the most straightforward way, **iframes** can be used. By using an `<iframe>` tag with a `src` attribute that points to a URL, one could embed visual components of one microfrontend (fragments) within another, without sacrificing on strong isolation between the two (Geers, 2020). The disadvantage here is the notoriously bad characteristics of iframes. One of these is bad performance: according to Souders (2013), iframes are up to 2 orders of magnitude more expensive to create than any other Document Object Model (DOM) element. Moreover, iframes also block the loading process of the rest of the page. Other suboptimal characteristics include accessibility and search engine optimization (SEO).

The simplicity of the web approach is also where it lacks applicability for larger projects. One of the biggest reasons to implement microservices and microfrontends is scalability, which the web approach is often not very suitable for.

Server-side composition

To accommodate for scalability and ensure higher performance, the composition of the microfrontends can be done before even reaching the end user's web browser. This will require dynamically composing the application on the server-side.

According to Geers (2020), this server-side composition can also provide a solid basis to be able to enable progressive enhancement, and is a big advantage for SEO.

There are multiple techniques to enable server-side composition. These include, but are not limited to:

- **Server-Side Includes (SSI)**.
  This uses an *SSI directive* (a HTML comment) to signify a placeholder for fragments to be rendered into.

```
<!-- #include virtual="/fragment" -->
```

It also supports the execution of commands from the server, and even some conditional logic (Apache, 2013). SSI has been around for a long time, this way most web servers have support for it. Because its directives are HTML comments, using these on a server that is not configured for SSI won't crash the application (the fragment will just simply not be rendered).

- **Edge-Side Includes (ESI)**.
  Instead of HTML comments, ESI uses XML-based ESI tags:

```
<esi:include  src="/fragment"
              alt="/fallback"
              onerror="continue" />
```

  ESI has a more extensive amount of functionalities (error handling, fallbacks, ...) compared to SSI. It is however more difficult to implement, because while SSI can work for nearly all web servers, ESI requires more specialized set-ups that are more complex to configure (Rappl, 2021).

Client-side composition

While with server-side composition, a web server was used as an aggregation layer for integrating the microfrontends; with client-side composition, the goal is to put this responsibility onto the end user's browser itself. The browser will have to render one parent HTML document which contains the instructions to integrate all the individual microfrontends into itself.

This parent HTML document is often called the **application shell** (Geers, 2020)(Rappl, 2021).

The microfrontend scripts and resources are often served from a **Content Delivery Network (CDN)**.

Client-side composition can be done in various different ways. A selection is outlined below:

- **Web Components**
  According to Mozilla (2021b), this term encompasses three main technologies, namely custom elements, shadow DOM (that can provide reusable DOM trees to ensure isolation) and HTML templates.
- **Single-page application (SPA) composition**
  In recent years, client-side frameworks have become the staple of fast and app-like experiences on the web. Most of these frameworks introduce a custom client-side routing solution. When using SPA composition, the application shell needs to take responsibility for the routing. The application shell will now not just render fragments using HTML, but execute scripts defined in the microfrontends themselves to be able to integrate them.

Client-side composition is great for delivering highly interactive, dynamic web applications. However, because the application is not served to the browser in full, the downsides include suboptimal SEO and an increased time to first load.

Universal composition

Aiming to combine the client-side and server-side composition approaches, and get the advantage of both, a hybrid approach can be achieved. Universal – or *isomorphic* – composition describes the process of

1. composing the application on the server-side, providing a fast initial load and thus enabling progressive enhancement and optimal SEO
2. *hydrating* the application so it becomes fully client-side interactive, providing the benefits of a highly dynamic application.

2.3.4 Usage of microfrontends

Microfrontends are being used by companies all over the world. Swedish furniture company IKEA mainly uses autonomous vertical teams that can develop in different technologies if necessary (Stenberg, 2018). Spotify, a music streaming service provider from that same country, uses iframe-based microfrontends within its desktop application to be able to develop different parts from the same view independently. These so-called *Spotlets* are developed by *Squads*, independent cross-functional teams (Gall, 2018). German e-commerce fashion retailer Zalando even created "Project Mosaic"[13], which contains a plethora of libraries and services to create both frontend and backend microservices.

2.3.5 Benefits of microfrontends

The microfrontend architecture carries over a lot of the advantages of the microservice architecture (Jackson, 2019): **technology independence** is now also possible across different frontend teams, allowing them to select the best tools and frameworks for the job. Loose coupling enables the **scalibility** of both the frontend and backend. Now, the frontend can also enjoy the benefits of independent deploys, isolated risks, and smaller codebases.

Teams also benefit greatly and can be reorganized to even greater benefit, as was discussed in detail in section 2.3.2 ("*Feature teams*").

These factors can have great results for a business. Due to their loose coupling, isolated features, and independent deployments, microfrontends can have vastly different release cycles, and iteration and **feature development** is usually faster because the teams don't have to wait for each other (Geers, 2020). Rappl (2021), because there is an existing composition mechanism, different microfrontends can be served to different users, enabling the introduction of **A/B testing** without significant changes in the microfrontends themselves.

[13] https://www.mosaic9.org/

2.3.6 Downsides and challenges of microfrontends

However, the microfrontend architecture does come with a significant amount of tradeoffs that need to be considered.

To cite Cam Jackson (2019):

> *"There are no free lunches when it comes to software architecture - everything comes with a cost."*

Organizational complexity might be one of the biggest downsides of the microfrontend approach. This makes microfrontends harder to recommend to small development teams and companies. **Domain decoposition** is often difficult and relies on a lot of organizational and technical factors.

Operational complexity also increases: debugging, logging, monitoring... all get a lot more complicated. Communication between the different microfrontends is generally also way more complex. If the microfrontend solution introduces the need for an application shell, this needs to be closely managed since it introduces a new single point of failure.

Lastly, lots of challenges are inherited from the microservice architecture, as described in *"Challenges and limitations of the microservice architecture"*.

2.4 Blazor WebAssembly

On the 6th of February 2018, Daniel Roth – Program Manager on the ASP.NET team at Microsoft – released a blog post called *A new experiment: Browser-based web apps with .NET and Blazor*. In this post, Roth (2018) announces an experimental project from the ASP.NET team: a component-oriënted web UI framework based on C#, .NET, HTML and so-called Razor pages.

The promise that was outlined by this post was a way to enable developers to write web applications using .NET technologies, rather than resorting to Javascript[14], the primary scripting language used on the web.

Executing .NET binaries within a web browser is made possible by **WebAssembly**[15], a binary instruction format. WebAssembly has been added to the World Wide Web Consortium (W3C) recommendation list, and has become the fourth language to run natively in web browsers, alongside HTML, CSS and Javascript (Couriol, 2019).

An overview of how Blazor works is provided in Figure 2.5.

Rather than transpiling every .NET assembly to WebAssembly, or relying on plugins, Blazor just relies on a .NET runtime that can run inside the browser sandbox, just like regular Javascript does. The current implementation of Blazor uses the WebAssembly-compiled

[14] Javascript is an implementation of the ECMAScript specification. Read more on https://ecma-international.org/tc39

[15] https://webassembly.org/

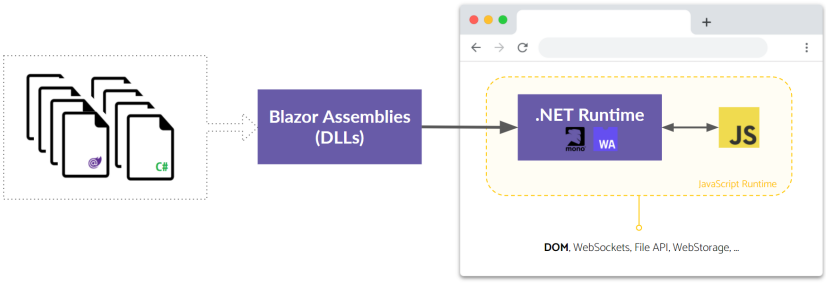

Figure 2.5: A diagram that outlines how Blazor WebAssembly works. The C# code and Razor files get compiled into IL assemblies, and run on a WebAssembly version of the Mono platform – a .NET runtime – inside the browser sandbox.

version of the Mono[16] platform – an open source .NET runtime – as an intermediate language (IL) interpreter to execute managed code at runtime.

Because of this, applications can leverage all standard web technologies like websockets, the DOM, and all other browser APIs, via **Javascript interoperability (JS interop)**. It also ensures the various security protections put in place by the sandbox environment to prevent malicious client-side attacks.

2.4.1 Current state of Blazor and microfrontends

According to Rappl (2020), an easy and effective way to make distributed development possible in a Blazor WebAssembly project, is simply to use separately distributed **component libraries**. In Blazor – and in the .NET ecosystem in general – NuGet[17] is the standardized way of library distribution. One could create an application shell which imports and incorporates these NuGet packages. While this implementation pattern makes development relatively straightforward, it has its downsides.

Because the integration happens at build-time; any change requires full recompilation of the entire application. Also, upon startup, the application shell has to have full knowledge of all libraries, and they all have to be loaded for the application to work, which makes this approach suboptimal regarding scalability. This integration method re-introduces coupling, and is generally discouraged (Jackson, 2019).

Integrating **Blazor components in a JavaScript-based app shell** is another option. While this has been done succesfully[18], and there are already some frameworks that support

[16] https://mono-project.com/
[17] https://www.nuget.org
[18] See an example here: https://github.com/lauchacarro/MicroFrontend-Blazor-React

this idea[19], it is not within the scope of this thesis, as the focus is on an almost exclusive .NET-approach.

Challenges and potential solutions

When looking for an approach that can dynamically load assemblies at runtime, the client-side **routing** starts to present a problem. When defining a page component in Blazor, the `@page` directive can be used. This will later provide the generated class with a `RouteAttribute` with a value that indicates the component's desired route template. The standard Blazor router will then use reflection on the specified `AppAssembly` to scan the loaded assemblies for these `RouteAttributes` (Sainty, 2019). This starts to become an issue if the assemblies are dynamically loaded – and thus not present upon compilation.

Another challenge that can present itself is **debugging**. When running Blazor DLLs on a WebAssembly runtime, a mechanism is needed to provide the link between the browser and the debugging tools: the debugging proxy. According to Abdalla (2020), this is a separate process that gets launched to load so-called program database (PDB) files. These are also called *symbol* files, and they are the link between the debugger and the source code (Microsoft, 2021). Loading the microfrontend symbol files dynamically is a challenge that would need to be overcome to allow the debugging of Blazor applications using microfrontends.

These are challenges that would not come up when the composition would be done at build-time like with component libraries. Luckily, the .NET 5 version of Blazor introduces lazy-loading capabilities out of the box[20], so most of these challenges that come from the dynamic loading requirement can most likely be overcome. According to Kdouh (2020), the Blazor router component, for example, now supports passing it additional assemblies to consider. For debugging, the `AssemblyLoadContext` can support the dynamic loading of dependencies with their symbols (Microsoft, 2019).

[19] For example the Piral microfrontend framework with their `piral-blazor` converter. See https://piral.io
[20] https://docs.microsoft.com/en-us/aspnet/core/blazor/webassembly-lazy- load-assemblies

3. Methodology

The conducted research within this thesis can be subdivided into 2 parts, as to accommodate the research objective:

- A theoretical study
- A proof of concept

3.1 Theoretical study

To be able to answer the research questions, a theoretical study was conducted in 4 phases:

Phase 1: Investigating what distributed development entails, and what benefits and challenges it brings within a company context.

Phase 2: Providing an overview of how the microfrontend architecture pattern came to fruition historically, and laying out the key differences with other more monolithic architectures.

Phase 3: Exploring characteristics, implementation patterns, benefits, challenges and best practices of the microfrontend architecture pattern.

Phase 4: Investigating the usage of the microfrontend architecture pattern in a Blazor WebAssembly project, and also looking at the possible challenges and their probable solutions.

The existing literature was scoured in the form of academic papers, conferences, books, blog posts and talks from renowned technology evangelists, etc...

The results of this theoretical study were incorporated into Chapter 2 ("*State of the art*").

3.2 Proof of concept

To be able to answer the research objectives, a proof-of-concept application was created using various online and offline resources to aid in the conceptualization and development process. These resources included academic papers, books and blog posts; as well as video presentations and conference talks. Even open-source code found on GitHub[1] was used to find good solutions for the problems at hand.

Some of these concepts were tested separately in a small "dummy" application before being incorporated into the main solution, to ensure their robustness.

The development process and results of the proof-of-concept application can be found in Chapter 4 ("*Proof of concept*").

[1] Code sources include the ASP.NET Core repo found on https://github.com/dotnet/aspnetcore and the Piral.Blazor repo found on https://github.com/smapiot/Piral.Blazor

4. Proof of concept

In this chapter, a proof-of-concept solution around a specific business case is created to demonstrate the practical use of the microfrontend architecture pattern.

As was made apparent in section 2.3.3 ("*Common implementation patterns*"), there are several different ways of conceptualizing and creating microfrontend solutions. This proof of concept (PoC) will focus on **universal composition**, to be able to enable **progressive enhancement** of the resulting web application. The desired solution ideally uses an exclusive **.NET approach** (i.e. using exclusively technologies within the .NET ecosystem), and has **run-time integration** with dynamic (lazy) loading capabilities.

4.1 Domain

4.1.1 Description

This PoC is centered around an e-commerce domain. To make the business case more specific and tangible, the case was formulated as follows:

> The creation of a web application for an online store dedicated to selling tabletop games (board games, card games, ...).

This encompasses the following functionalities:

- Browse games
- View game details
- Order games

This domain was chosen because the technical needs of e-commerce solutions tend to

overlap with the set-out desired characteristics of the proof-of-concept application. In a real-world scenario, the reverse process would be necessary, and one would have to pick the best suiting implementation pattern for the case at hand.

The universal rendering will make sure the application has a faster initial load and better SEO than a purely client-side rendered application, while maintaining a highly dynamic nature.

4.1.2 Decomposition into microfrontends

According to Rappl (2021), the domain decomposition strategy is a crucial foundation for determining the architecture of the microfrontend solution. In a real company context, both technical and organizational factors need to be considered to be able to split up the domain. For this PoC, the following modules were chosen:

- The *Discover* microfrontend, containing the product overviews and product details.
- The *Order* microfrontend, containing everything necessary for ordering products.

4.2 Architecture

Using the domain decomposition results, the overall architecture of the solution can be determined. An overview is given in Figure 4.1. It mainly consists of 3 parts:

- The *Discover* microfrontend
- The *Order* microfrontend
- The application shell where these microfrontends will get integrated into.

4.2.1 Composition structure

A visual overview for the component and page composition for the PoC application is shown in Figure 4.2. Pages are registered on a certain route and fragments are registered into a slot (with a unique name) in another microfrontend or the application shell.

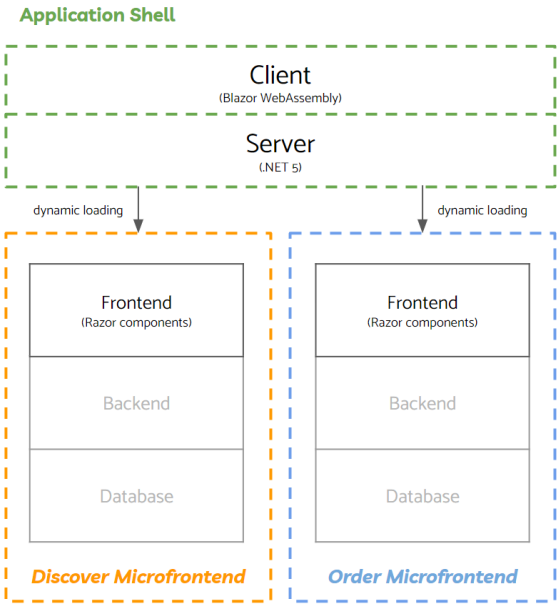

Figure 4.1: The architecture overview for the proof-of-concept solution. In the microfrontends, only the frontend has been implemented in the PoC, while the other elements were mocked using in-memory solutions for demonstration purposes.

A summary of the components each microfrontend exposes is shown below:

Discover microfrontend:

Pages
- `GameDetails` with route `/game?id={id}`

Fragments
- `GameOverview` for slot `homepage`

Order microfrontend:

Pages
- `CartOverview` with route `/cart`
- `OrderConfirmation` with route `/orderconfirmation`

Fragments
- `AddToCartButton` for slot `game-actions`
- `CartLink` for slot `top-row-items`

4.3 Development

In what follows, more information about the development of the PoC solution will be given. This explanation is accompanied by code that can be found at

https://github.com/DanteDeRuwe/bachelor-thesis-code.

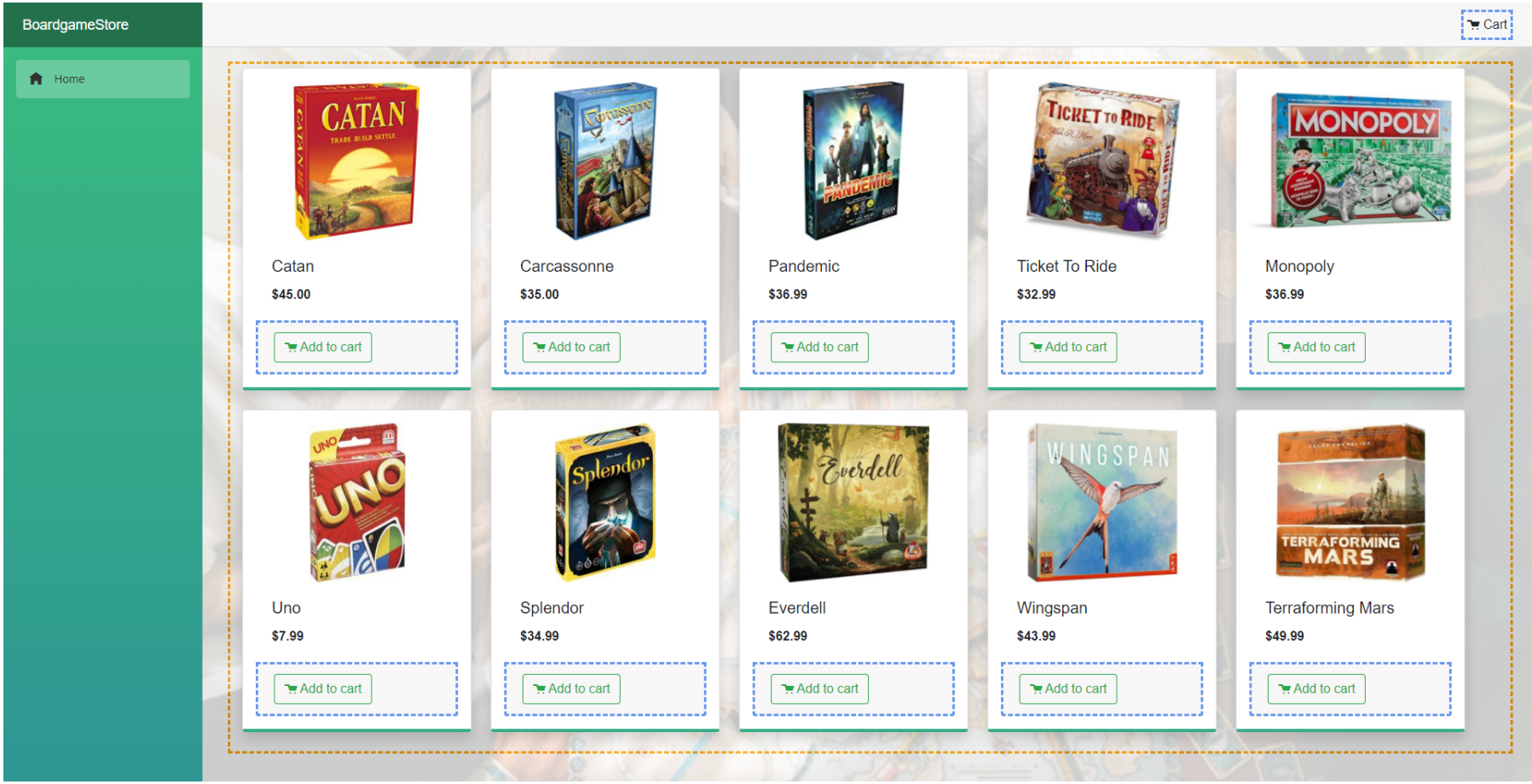

Figure 4.2: A visual overview for the component and page composition for the PoC application. The colored dashed lines indicate the sources of the components (blue for *"Order"* microfrontend, orange for *"Discover"*).

4.3.1 Preparation

An ASP.NET hosted Blazor WebAssembly project was created. This can be done via the .NET Command Line Interface (CLI):

```
$ dotnet new blazorwasm --hosted
```

This solution will serve as the application shell of the microfrontend application. It consists of a client and a server part.

Next, pre-rendering was enabled in the application shell, using a strategy outlined in a blog post by Jon Hilton [1].

4.3.2 Process and results

The framework library

To be able to dynamically integrate the microfrontends into the application shell at runtime, it is useful to create a baseline framework with various utilities and reusable components.

This is the goal of the `MicrofrontendFramework.Blazor`[2] project. It has the following functionalities:

- Fragment rendering: the framework exposes a `[Fragment]` attribute to register a component as a fragment with a custom name. It then also exposes a `<Fragment/>` component that is used from the application shell or other microfrontends to render a fragment by name.
- Routing: the framework exposes a `<DynamicRouter/>` component that can be used instead of the default Blazor router to render the corresponding component dynamically
- Component registration and dependency injection (DI): This functionality is exposed as via an extension method called `AddMicroFrontends` that, given a collection of loaded assemblies, can extract a collection of all components, and register this collection onto the DI container. It also will register the services that these microfrontends need onto the provided service collection.[3]

The application shell

The main responsibility of the application shell is to compose the microfrontend components and integrate the microfrontends into itself. This includes executing the functionalities present in the framework library such as **routing** and **fragment rendering**. Before it can make use of the `AddMicroFrontends` extension method, it should dynamically load the microfrontend assemblies to be able to pass these as an argument.

[1] https://jonhilton.net/blazor-wasm-prerendering/

[2] https://github.com/DanteDeRuwe/bachelor-thesis-code/tree/main/src/framework/MicrofrontendFramework.Blazor

[3] It does this by looking up a `Microfrontend` class in the microfrontend and reflectively calling its `Configure` method. See MicrofrontendFramework.Blazor/Extensions.cs:30-39

On the server-side, as by way of proof of concept, the application shell fetches the microfrontend assemblies from disk, acting as an ad hoc CDN. The server then also exposes the file paths via a RESTful API.[4] From the client-side, the file paths are fetched via an HTTP call to the server API, and the files themselves are then fetched via their paths.

The assemblies are stored in dynamic-link library (DLL) files. To also enable the debugging of the microfrontends, another type of file is also needed: the PDB files. As mentioned before, these are also called *symbol* files, and they are the link between the debugger and the source code (Microsoft, 2021).

On both the server and the client side, each assembly and symbol definition can be loaded in memory using the default `AssemblyLoadContext`:

```
AssemblyLoadContext.Default.LoadFromStream(dllStream, pdbStream);
```

This will allow the debugger to communicate with the browser at runtime, and makes sure the standard **debugging experience** that Blazor developers expect will work on this microfrontend solution.

Once the assemblies are loaded, they can be passed on to the aforementioned `AddMicroFrontends` extension method originating from the framework project.

Next to fragment rendering, dynamic loading, and routing, the application shell also has responsibility for some **shared capabilities** such as navigation, and basic **layout**.

The microfrontends

In this PoC solution, every microfrontend is just a Razor class library exposing a certain set of pages and fragments. An overview of these exposed elements was shown in 4.2.1 ("*Composition structure*"). These exposed components are then picked up by the application shell (using the framework library) for integration.

Each microfrontend is also an independent owner of its own data. They are not strongly coupled to the application shell in any way, and only use the framework library to be able to expose fragments and use placeholder slots for for the application shell to integrate other fragments into.

[4] In a real-world scenario, these responsibilities would probably be assigned to a dedicated service.

5. Discussion

The microfrontend architecture can provide development teams with significant benefits if they are willing to accept the drawbacks and combat the challenges that go along with its implementation.

The modular character of this architecture pattern can, in theory, be one of the most effective ways to achieve loosely coupled end-to-end services (*"from database to UI"*), which can be developed and managed by autonomous cross-functional teams. This can enable development teams to be (geographically) distributed. Another benefit of operating autonomously is the reduced waiting time and faster release cycles for the individual modules.

However, this organizational shift might not be easy or even possible in some cases. Especially in smaller companies, the advantages and disadvantages should carefully be weighed.

On the technical side, creating a solid microfrontend solution has its challenges too. Creating a robust application all starts with a good conceptual and architectural baseline. This is no different when applying the microfrontend architecture to the Blazor WebAssembly technology. To be able to achieve very loose coupling of the different modules, a dynamic loading strategy of the needed assemblies can be necessary. This introduces the challenges of routing and fragment rendering, both of which were addressed by the proof-of-concept solution found in Chapter 4 ("*Proof of concept*").

Creating a real-world microfrontend solution within the .NET ecosystem using Blazor WebAssembly is possible, but like everything, can be improved. Further research and experimental work can be done on providing a way to enable every microfrontend to bring its own dependencies and static files, while still ensuring isolation and performance, and avoiding runtime conflicts. Also research in how to develop the microfrontends separately

from the full application shell can be helpful. One way of potentially achieving this is to bundle up the application shell into some kind of emulator to enable local development. This would also enable the local debugging of these microfrontends without the need for the full application shell.

Without much doubt, the future versions of Blazor will bring more possibilities to solve the problems and challenges that come up when developing microfrontends with Blazor. Performance keeps improving, and new ways of dynamic loading and server-side rendering are being introduced.

While some form of microfrontends have made their introduction in large companies recently, the architecture pattern is just slowly breaking into the *mainstream* development community. It will be interesting to see what solutions and frameworks will come up to make the development of these modular applications easier than ever before.

A. Proposal

This bachelor thesis is based on a bachelor thesis proposal that was evaluated by the promotor beforehand. The proposal can be found in the appendix below.

Enabling the Distributed Development of Blazor-Based Web Applications Using a Microfrontend Architecture

Bachelor Thesis Proposal 2020-2021

Dante De Ruwe[1]

Abstract
With the WebAssembly *(WASM)* standard a new set of web applications have been made possible. Within the .NET ecosystem, the most popular solution for generating WebAssembly applications is called Blazor. This framework allows building interactive web UIs using C# instead of JavaScript. One challenge in this approach is that there is no direct way of enabling distributed development. While component libraries can be created independently, knowledge in the main application would be required for integration. Using a *microfrontend architecture* this relationship could be inversed. This thesis investigates what is needed to empower the distributed development of large-scale Blazor-based web applications, tackling the individual challenges that go along with this on its way. To gain insight into the domain, a theoretical study will be conducted in 3 phases. Ultimately, a proof-of-concept solution will be created to demonstrate the use of a microfrontend architecture in a Blazor environment with an almost exclusive .NET-approach. The prediction is that this should be feasible, but well-thought-out compromises will have to be made regarding the limitations of the current state of the Blazor framework and WebAssembly standard. As the adoption of the Blazor framework and the microfrontend architecture pattern will mature, this thesis could be a valuable resource for .NET-focussed development teams creating large-scale full-stack web applications.

Keywords
Web Development – Software Architecture – Distributed Development – Microfrontends – Blazor – WebAssembly – .NET – C#

Co-promoter
Dr. Florian Rappl[2] (Solution Architect at smapiot GmbH)

Contact: [1]dantederuwe@gmail.com; [2]florian.rappl@smapiot.com

Contents

1. Introduction

The development of applications for the web has seen some dramatic shifts over the years. Apart from new technologies, protocols and standards, the way web applications are structured has undergone some evolutions as well. "Software architecture" not only outlines the pure structure of the application, but also defines the responsibility of all the pieces of the application, and how these pieces ought to interact with each other (Fedorov et al., 1998).

In the "early days" of web development, nearly all applications had a *monolithic* architecture: a single-tier architecture, where the user interface (UI), business logic and data storage are all managed in a single all-in-one solution. Now, developer teams have mostly adopted split-stack development, where the UI is handled in the so-called "frontend" and the business and data logic are dealt with by a "backend" system. This reduced coupling enabled specialized teams to develop each aspect individually, independently and therefore simultaneously (Dunkley, 2016). For the same reasons, *microservices* started making an appearance when developers realized that having a single backend service could also be considered a monolithic approach (Fowler & Lewis, 2014).

In 2016 the ThoughtWorks Technology Radar (ThoughtWorks, 2020) coined the term "*Micro Frontends*" to describe the split of the frontend monolith into independently deployable and maintainable pieces. This is especially beneficial for large-scale projects, where the division of developers

in autonomous teams is quite common. However, enabling distributed development of these microfrontends is not a trivial undertaking, and naturally has its challenges.

This thesis aims to be an application of the microfrontends architecture pattern, to enable distributed development of large-scale web applications. More specifically, this thesis will focus on the Blazor[1] web framework that uses WebAssembly[2] to execute C#[3] code in the browser.

The research questions that will be addressed:

- RQ_1 What is needed to be able to independently develop and deploy microfrontends using Blazor WASM?
- RQ_2 How to render Blazor-based microfrontends with the proper isolation, performance and with progressive enhancement[4] in mind?
- RQ_3 What are the challenges that need to be overcome, and how would one do so?
- RQ_4 How can development teams benefit from the transformation of their Blazor monolith into a microfrontend solution?

Additionally, an objective of this thesis is to generate an open-source proof-of-concept Blazor web application that implements the microfrontend architecture pattern.

2. State-of-the-art

2.1 Distributed Development

As software projects grow larger and larger in size, naturally more developers are needed to maintain them. But the efficiency of teams does not scale linearly. This is a wisdom that dates back to the very early days of software development, as proven by the still applicable quote by Brooks (1975): “adding manpower to a late software project makes it later”.
A solution to this is obviously to subdivide the project into chunks, managed by specialized teams. A company could even let these teams operate from different geographical locations. This is called “distributed development”. While there are major benefits; this significantly increases the need for extensive coordination and communication between the teams, as well as a good company structure and strategy. Making this possible all starts with the choice of a good common architecture that is scalable and supports distributed development (Yuhong, 2008).

2.2 Why Microfrontends?

As outlined in the introduction to this proposal, there is an already widespread practice of splitting up projects “horizontally” per layer or technology. It makes sure that experts can co-operate together as one team and ensure a high technical standard within the boundaries of their respective areas of expertise.

But what if a company wants to put its focus on user experience, innovation and features, instead of purely on creating the most technically perfect solution? This is where multi-disciplinary or cross-functional “feature teams” can come in. These are grouped around a specific business case or customer need. This “vertical slicing” has major benefits: it enables teams to be completely independent and have end-to-end responsibility for the features they develop. This aids in accelerating development speed, cutting down on inter-team communication and enabling developers to feel a greater sense of involvement in the project or product (Larman & Vodde, 2008).

The “microfrontends” architecture gives developers the necessary tools to be able to organize themselves into autonomous “feature teams”, where each team has a company mission they specialize in. (Geers, 2020)

2.3 Composing and Integrating Microfrontends

Implementing the microfrontend architecture pattern can be done in various ways. This proposal will not mention all of them, nor will it provide a detailed look at any of these techniques. However, below some common methodologies are outlined. (Geers, 2020) (Peltonen et al., 2020) (Pavlenko et al., 2020)

- The web approach (hyperlinks, iframes...)
- Server-side rendering
- Client-side rendering
- Universal rendering

This thesis will focus on universal (also known as *isomorphic*) rendering. This is a rendering technique that aims to combine client-side and server-side rendering practices. The main concept is pre-rendering the application on the server, resulting in a relatively fast initial load, and then *(re)hydrating* the pre-rendered application on the client side to make it fully client-side interactive. This enables progressive enhancement. (Miller & Osmani, 2019)

2.4 Blazor WebAssembly

According to Rappl (2020b), an easy and effective way to make distributed development possible in a Blazor WebAssembly project, is simply to use separately distributed component libraries. In Blazor – and in the .NET ecosystem in general – NuGet[5] is the standardized way of library distribution. One could create one overarching web app which imports and incorporates these NuGet packages. This is often called the *app shell* (Geers, 2020). While this implementation pattern makes development relatively straight-forward, it has its downsides. Because the integration happens at build-time; any change requires full recompilation of the entire application. Also, upon startup, all libraries have to be loaded, which makes this approach sub-optimal regarding scalability. This integration method re-introduces coupling, and is generally discouraged (Jackson, 2019).

Integrating Blazor components in a JavaScript based app shell is another option. While this has been done successfully[6], this is not within the scope of this thesis, as the focus is on an almost exclusive .NET-approach.

[1] https://blazor.net
[2] https://webassembly.org
[3] http://csharp.net
[4] https://wikipedia.org/wiki/Progressive_enhancement
[5] https://www.nuget.org
[6] See an example here: https://github.com/lauchacarro/MicroFrontend-Blazor-React

Enabling the Distributed Development of Blazor-Based Web Applications Using a Microfrontend Architecture

The desired solution, in contrast to the aforementioned approaches, uses *"C#.NET all the way"* if possible, and has run-time integration with dynamic (lazy) loading capabilities. In this, the challenges may come because of the Blazor Intermediate Language (IL) trimming, which removes unused code and libraries in the output assemblies (Latham, 2020). Luckily, .NET 5 introduces lazy loading capabilities for Blazor out of the box [7] (Kdouh, 2020) (Rappl, 2020a), yet it is unclear if this is the cure-all solution.

It bears repeating that progressive enhancement is desired. The combination of these requirements will be achieved by using universal rendering, but with the focus on the server-side.

3. Methodology

3.1 Theoretical study

To be able to answer the research questions, a theoretical study will be conducted in 3 phases:

- Phase 1: Providing an overview of the microfrontend architecture and the key differences with a monolithic architecture.
- Phase 2: Exploring implementation patterns, challenges and best practices of the microfrontends pattern in a Blazor WebAssembly project.
- Phase 3: Investigating the benefits and drawbacks of using the microfrontends architecture with the goal of enabling distributed development in a company context.

3.2 Proof of concept

With the gained insight of the literature and the theoretical study, a proof-of-concept solution around a realistic business case will be created to demonstrate the practical application of the microfrontends architecture pattern in a Blazor WebAssembly project. Ideally, this proof of concept will be made open-source.

4. Expected Results

The expected results of the theoretical study are relatively straightforward. In *phase 1*, further insight will be gained into the microfrontend architecture pattern. *Phase 2* is predicted to provide valuable knowledge for the eventual proof-of-concept solution. In particular, the challenges that will undoubtedly come up in this phase, will be important considerations. Some expected challenges include debugging, assembly sharing, dynamic loading in combination with IL trimming (≈ *tree shaking*), etc... With the results of research *phase 3*, insight can be gained into the cost-benefit balance of the microfrontends approach; and indicate when and for whom it is appropriate.

The proof of concept should be feasible to create, but well-thought-out compromises will have to be made regarding the limitations of the current state of the Blazor framework and WebAssembly standard.

[7] https://docs.microsoft.com/en-us/aspnet/core/blazor/webassembly-lazy-load-assemblies

5. Expected Conclusions

The distributed development of Blazor-based web applications, using the microfrontends pattern, with an almost exclusive .NET-approach should be possible. Because of the complexity of integration, this is only beneficial to large teams, or teams that expect scaling. Judging by the limited maturity of Blazor and the application of the microfrontends pattern within this tech stack, this thesis could provide one of the first realistic business cases within this specific area.

References

Brooks, F. P. (1975). *The mythical man-month*. Addison-Wesley Publishing Company.

Dunkley, W. (2016, June 7). *Split stack development: A model for modern applications*. Retrieved November 30, 2020, from https://medium.com/@Future_-_Friendly/split-stack-development-a-model-for-modern-applications-d7b9abb47bd5

Fedorov, A., Francis, B., Harrison, R., Murphy, S., Smith, R., Sussman, D., & Wood, S. (1998). *Professional active server pages 2.0*. Wrox Press. Retrieved November 30, 2020, from https://docs.microsoft.com/en-us/previous-versions/office/developer/server-technologies/aa480455(v=msdn.10)

Fowler, M., & Lewis, J. (2014, March 25). *Microservices*. Retrieved November 30, 2020, from https://martinfowler.com/articles/microservices.html

Geers, M. (2020). *Micro frontends in action*. Manning Publications.

Jackson, C. (2019, June 19). *Micro frontends*. Retrieved November 30, 2020, from https://martinfowler.com/articles/micro-frontends.html

Kdouh, W. (2020, December 23). *Microfrontends with blazor webassembly*. Retrieved December 26, 2020, from https://medium.com/@waelkdouh/microfrontends-with-blazor-webassembly-b25e4ba3f325

Larman, C., & Vodde, B. (2008). Feature teams. *Scaling lean & agile development: Thinking and organizational tools for large-scale scrum* (pp. 150–192). Addison-Wesley Professional.

Latham, L. (2020, May 19). *Configure the linker for asp.net core blazor*. Retrieved December 24, 2020, from https://docs.microsoft.com/en-us/aspnet/core/blazor/host-and-deploy/configure-linker?view=aspnetcore-3.1

Miller, J., & Osmani, A. (2019). *Rendering on the web*. Retrieved December 15, 2020, from https://developers.google.com/web/updates/2019/02/rendering-on-the-web#rehydration

Pavlenko, A., Askarbekuly, N., Megha, S., & Mazzara, M. (2020). Micro-frontends: Application of microservices to web front-ends. https://doi.org/10.22667/JISIS.2020.05.31.049

Peltonen, S., Mezzalira, L., & Taibi, D. (2020). Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. Retrieved December 15, 2020, from https://arxiv.org/pdf/2007.00293.pdf

Rappl, F. (2020a, November 12). *Microfrontends with blazor: Welcome to the party! (.NET Conf 2020)*. Retrieved November 28, 2020, from https://youtu.be/npff2NjVXEE

Rappl, F. (2020b, May 20). *Microfrontends with blazor: Welcome to the party! (München .NET Meetup 2020)*. Retrieved December 16, 2020, from https://youtu.be/IGVQoCzCtR4

ThoughtWorks. (2020). *Micro frontends technology radar*. Retrieved November 30, 2020, from https://thoughtworks.com/radar/techniques/micro-frontends

Yuhong, Y. (2008). Geographically distributed development : Trends, challenges and best practices. Retrieved December 1, 2020, from https://dspace.mit.edu/bitstream/handle/1721.1/42373/234383465-MIT.pdf

Bibliography

Aarsten, A., Brugali, D., & Menga, G. (1996). Patterns for three-tier client/server applications. *Proceedings of Pattern Languages of Programs (PLoP'96)*, *4*.

Abdalla, S. (2020, November 10). Under the hood with debugging in blazor webassembly. Retrieved August 21, 2021, from http://blog.safia.rocks/blazor-wasm-debugging

Apache. (2013). Retrieved August 20, 2021, from http://httpd.apache.org/docs/current/howto/ssi.html

Avraham, S. B. (2017, September 5). What is rest — a simple explanation for beginners, part 1: Introduction. Retrieved August 26, 2021, from https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f

Ball, K. (2019, January 30). *The increasing nature of frontend complexity*. Retrieved April 17, 2021, from https://blog.logrocket.com/the-increasing-nature-of-frontend-complexity-b73c784c09ae/

Carmel, E., Espinosa, J. A., & Dubinsky, Y. (2010). "follow the sun" workflow in global software development. *Journal of Management Information Systems*, *27*(1), 17–38. http://fs2.american.edu/alberto/www/papers/JMIS_2010_CarmelEspinosaDubinski_FollowTheSun.pdf

Conchúir, E. Ó., Ågerfalk, P. J., Olsson, H. H., & Fitzgerald, B. (2009). Global software development: Where are the benefits? *Communications of the ACM*, *52*(8), 127–131.

Couriol, B. (2019, December 6). *Webassembly 1.0 becomes a w3c recommendation and the fourth language to run natively in browsers*. Retrieved May 1, 2021, from https://www.infoq.com/news/2019/12/webassembly-w3c-recommendation/

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. *Present and*

ulterior software engineering (pp. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12

Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., & Safina, L. (2018). Microservices: How to make your application scale. In A. K. Petrenko & A. Voronkov (Eds.), *Perspectives of system informatics* (pp. 95–104). Springer International Publishing. https://doi.org/https://doi.org/10.1007/978-3-319-74313-4_8

Dunkley, W. (2016, June 7). *Split stack development: A model for modern applications*. Retrieved November 30, 2020, from https://medium.com/@Future__Friendly/split-stack-development-a-model-for-modern-applications-d7b9abb47bd5

Education, I. C. (2021). Soa. Retrieved August 26, 2021, from https://www.ibm.com/cloud/learn/soa

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

Fedorov, A., Francis, B., Harrison, R., Murphy, S., Smith, R., Sussman, D., & Wood, S. (1998). *Professional active server pages 2.0*. Wrox Press.

Fenton, S. (2012). Compiling vs transpiling. Retrieved August 26, 2021, from https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *Hypertext transfer protocol – HTTP/1.1* (tech. rep.). RFC Editor. https://doi.org/10.17487/rfc2616

Fowler, M., & Lewis, J. (2014, March 25). *Microservices*. Retrieved November 30, 2020, from https://martinfowler.com/articles/microservices.html

Gall, R. (2018, July 19). *Spotify has "one of the most intricate uses of javascript in the world," says former engineer | packt hub*. Retrieved August 20, 2021, from https://hub.packtpub.com/spotify-has-one-of-the-most-intricate-uses-of-javascript-in-the-world-says-former-engineer/

Gallaugher, J. M., & Ramanathan, S. C. (1996). Choosing a client/server architecture. *Information Systems Management*, *13*(2), 7–13. https://doi.org/10.1080/10580539608906981

Geers, M. (2020). *Micro frontends in action*. Manning Publications.

Gysels, N. (2020). Hoe implementeert men een ecosysteem van microservices bij applicatieontwikkeling? https://catalogus.hogent.be/catalog/hog01:000733123

Jackson, C. (2019, June 19). *Micro frontends*. Retrieved November 30, 2020, from https://martinfowler.com/articles/micro-frontends.html

Kdouh, W. (2020, December 23). *Microfrontends with blazor webassembly*. Retrieved December 26, 2020, from https://medium.com/@waelkdouh/microfrontends-with-blazor-webassembly-b25e4ba3f325

Kiel, L. (2003). Experiences in distributed development: A case study. *Proceedings of International Workshop on Global Software Development*, 44–47. http://www.gsd2003.cs.uvic.ca/gsd2003proceedings.pdf#page=46

Koren, I., & Klamma, R. (2018). The exploitation of openapi documentation for the generation of web frontends. *Companion Proceedings of the The Web Conference 2018*, 781–787. https://doi.org/10.1145/3184558.3188740

Larman, C., & Vodde, B. (2008). Feature teams. *Scaling lean & agile development: Thinking and organizational tools for large-scale scrum* (pp. 150–192). Addison-Wesley Professional.

Leff, A., & Rayfield, J. T. (2001). Web-application development using the model/view/controller design pattern. *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, 118–127. https://doi.org/10.1109/EDOC.2001.950428

Microsoft. (2016, June 20). What is managed code? Retrieved August 26, 2021, from https://docs.microsoft.com/en-us/dotnet/standard/managed-code#intermediate-language--execution

Microsoft. (2018, May 9). What is a content delivery network (cdn)? - azure. Retrieved August 26, 2021, from https://docs.microsoft.com/en-us/azure/cdn/cdn-overview

Microsoft. (2019, August 9). Understanding assemblyloadcontext - .net core. Retrieved August 21, 2021, from https://docs.microsoft.com/en-us/dotnet/core/dependency-loading/understanding-assemblyloadcontext#how-does-assemblyloadcontext-support-dynamic-dependencies

Microsoft. (2021, March 31). *Set symbol (.pdb) and source files in the debugger - visual studio (windows)*. Retrieved August 18, 2021, from https://docs.microsoft.com/en-us/visualstudio/debugger/specify-symbol-dot-pdb-and-source-files-in-the-visual-studio-debugger?view=vs-2019

Mozilla. (2021a). Retrieved August 26, 2021, from https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

Mozilla. (2021b). Retrieved August 20, 2021, from https://developer.mozilla.org/en-US/docs/Web/Web_Components

Newman, S. (2015). *Building microservices: Designing fine-grained systems*. " O'Reilly Media, Inc."

NGINX. (2021). Api gateway. Retrieved August 26, 2021, from https://www.nginx.com/learn/api-gateway/

Oshri, I., Kotlarsky, J., & Willcocks, L. P. (2015). *The handbook of global outsourcing and offshoring 3rd edition*. Springer.

Pavlenko, A., Askarbekuly, N., Megha, S., & Mazzara, M. (2020). Micro-frontends: Application of microservices to web front-ends. https://doi.org/10.22667/JISIS.2020.05.31.049

Rappl, F. (2019, February 25). *Taming the front-end monolith*. Retrieved April 17, 2021, from https://blog.logrocket.com/taming-the-front-end-monolith-dbaede402c39/

Rappl, F. (2020, May 20). *Microfrontends with blazor: Welcome to the party! (München .NET Meetup 2020)*. Retrieved December 16, 2020, from https://youtu.be/IGVQoCzCtR4

Rappl, F. (2021). *The art of micro frontends*. Packt Publishing Ltd.

Reese, G., & Oram, A. (2000). Distributed application architecture. *Database programming with jdbc and java* (pp. 126–145). O'Reilly. https://web.archive.org/web/20110406121920/http://java.sun.com/developer/Books/jdbc/ch07.pdf

Roth, D. (2018, February 6). *A new experiment: Browser-based web apps with .net and blazor*. Retrieved April 29, 2021, from https://devblogs.microsoft.com/aspnet/blazor-experimental-project/

Sainty, C. (2019, October 22). *Introduction to routing in blazor*. Retrieved August 21, 2021, from https://chrissainty.com/introduction-to-routing-in-blazor/

Sengupta, B., Chandra, S., & Sinha, V. (2006). A research agenda for distributed software development. *Proceedings of the 28th international conference on Software engi-*

neering, 731–740. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.626.5687&rep=rep1&type=pdf

Šmite, D., Moe, N. B., & Ågerfalk, P. J. (Eds.). (2010). *Agility across time and space: Implementing agile methods in global software projects*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-12442-6

Soldani, J., Tamburri, D. A., & Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, *146*, 215–232. https://doi.org/https://doi.org/10.1016/j.jss.2018.09.082

Souders, S. (2013). Retrieved August 9, 2021, from https://www.stevesouders.com/blog/2009/06/03/using-iframes-sparingly/

Stenberg, J. (2018, August 7). Retrieved August 20, 2021, from https://www.infoq.com/news/2018/08/experiences-micro-frontends/

Techopedia. (2012). Load balancer. Retrieved August 26, 2021, from https://www.techopedia.com/definition/29118/load-balancer

ThoughtWorks. (2020). *Micro frontends technology radar*. Retrieved November 30, 2020, from https://thoughtworks.com/radar/techniques/micro-frontends

W3.org. (2021). About w3c. Retrieved August 26, 2021, from https://www.w3.org/Consortium/

Webassembly.org. (2021). *Webassembly*. Retrieved May 1, 2021, from https://webassembly.org/

Yuhong, Y. (2008). Geographically distributed development : Trends, challenges and best practices. Retrieved December 1, 2020, from https://dspace.mit.edu/bitstream/handle/1721.1/42373/234383465-MIT.pdf