INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Evolução de um frontend monolítico na i2S: um caso de estudo

DIOGO RAMIRES MOTA LEITE
Outubro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Monolithic frontend evolution at i2S: a case study

Diogo Ramires da Mota Leite

Master's dissertation in
Informatics Engineering, Area of Specialization in
Software Engineering

Advisor: Isabel de Fátima Silva Azevedo
External advisor: Nuno Ferreira

Porto, September 2020

Dedication

This thesis is dedicated to my godson.
In the future, I wish to be a role model for him and with this thesis, I want to demonstrate that with hard work and focus he can accomplish everything that he wants.

Resumo

Enquanto se adaptam às demandas do mercado tecnológico e do mercado de negócios, as empresas necessitam de continuar a garantir uma resposta às necessidades dos seus clientes através do aumento dos seus padrões de qualidade, como por exemplo, usabilidade, confiabilidade, desempenho e suporte.

Abordagens relacionadas com novas arquiteturas de software associadas ao *Frontend* têm vindo a crescer de modo exponencial na última década.

Com *Micro Frontends* uma aplicação *web* é composta por várias peças isoladas, que são mantidas por diferentes equipas e que podem até operar em áreas de negócio diferentes. *Micro Frontends* é uma tendência ainda pouco explorada.

O principal objetivo desta tese é descrever um caso de estudo onde serão demonstrados os procedimentos necessários para ajustar um *frontend* monolítico ao conceito de *Micro Frontends*, considerando nesse processo as melhores práticas e padrões de engenharia. Para além disto, é necessário também avaliar e estimar os impactos e possíveis melhorias que esta abordagem traz. É importante também referir que a aplicação na qual esta abordagem será introduzida, é uma aplicação que durante o decorrer desta tese passou a ser utilizada em alguns clientes como Aplicação Beta. Por essa razão, as necessidades enunciadas anteriormente ganharam uma maior relevância devido à importância de responder às mesmas de forma rápida e adequada.

Contudo, o trabalho aqui documentado, não é sobre a migração em si, mas sim sobre a apresentação de um caso de estudo onde é apresentada uma possível solução para esta migração e onde são detalhados todos os passos e procedimentos necessários.

Palavras-chave: *Micro Frontends, Frontend* Monólitico, Angular, React, Evolução Arquitetural.

Abstract

Companies need to ensure a response to the clients needs with a constant increase in their standards, like usability, reliability, performance, and support, while adapting to technology and business market demands. Software architecture approaches to Frontend have been changing over the years and, particularly in the last decade, exponentially. With Micro Frontends, a web application is composed of pieces maintained by autonomous teams. They can be very different and even operate in distinct business domains. Micro Frontends is a tendency still underexplored.

This thesis's main objective is to describe a case study where it is demonstrated how to adjust a monolithic frontend to the concept of Micro Frontends, considering in this process the best engineering practices and standards. Besides, an assessment of its impacts and the real improvements need to be estimated. It is pertinent to note that the application where this change will occur is an application that, in the elapse of this thesis, started to be used in some clients as a Beta Application. Thus, the needs mentioned before have gained greater prominence with an imperative of adequate and rapid response.

However, the work documented here is not about the migration itself, but about presenting a case study where a possible solution for this migration is presented and where all the necessary steps and procedures are detailed.

Keywords: Micro Frontends, Monolithic Frontend, Angular, React, Architectural evolution.

Acknowledgments

I wish to express my gratitude to my external advisor and professor, Nuno Ferreira. Without his help, coach, and inspiration, nothing would have been done.

During this entire time, he played different roles in this thesis. The first role was as an external advisor and team leader, helping me with the business drivers and being always available to give some of his time to handle thesis problems.

The second role was as an ISEP professor, guiding me to produce a valuable thesis document. The last role was as a friend and co-worker every time that he spoke to me intending to give me motivation and push me to think by myself.

I also would like to express my gratitude to my advisor, Professor Isabel Azevedo, whose experience, good advice, knowledge, and patience, helped me to develop a thesis document that I am proud of.

Last but not least, I would like to demonstrate sincere thanks to two groups of people.

One group is my teammates, especially Hugo Oliveira, João Ribeiro, André Dias and Francisco Lima, whose advice, motivations, and expertise contributed on a large scale to this thesis and to my personal growth.

The second group is my friends from Portugal, Belgium, and Spain, and, in particular, my family and godson. They gave me emotional support the entire time and helped to manage my schedule, allowing me to separate the time that I should be focused on my work from the time that I should take to myself for talking and being with them.

Table of Contents

List of Figures

List of Tables

List of Source Code

Acronyms

API	Application programming interface
ATAM	Architecture Trade-off Analysis Method
AWS	Amazon Web Services
BFF	Backend for Frontend
CBAM	Cost-Benefit Analysis Method
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Mark-up Language
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
UI	User Interface
UML	Unified Modelling Language
UX	User Experience
XML	eXtensible Mark-up Language

1 Introduction

This document constitutes the final delivery of the master's thesis in Informatics Engineering at the Instituto Superior de Engenharia do Porto (ISEP). This work deals with the understanding of the concept of micro frontends and its possible use in a certain application developed by i2S, a company that develops insurance software. In this chapter, it is given a context for this thesis and it is described the problem that must be solved. Also, in here, it is provided the objectives related to this work and the development process that will be followed.

1.1 Context

The presented work is related to a Micro Frontend approach with an API Gateway demonstrated using i2S software. This company, with over 35 years of experience, develops software products for the insurance business. Those years of experience is one of the reasons why i2S wants to follow the micro frontends approach. The company to create the basis for sustained future front-end developments directed to fulfilling their clients needs - the way to respond to this demand is to keep up with technological developments. The insurance sector has passed by extensive reformulation and customer expectations have been transformed. Insurance company customers are digitally savvy and the demand for new digital channels and way of working with the insurance company rise, transferring the pressure to the insurance companies suppliers, like i2S.

This sector has been hesitant to keep pace with digitalization and is only now starting to understand the power of becoming more focus on customer needs and attempting to increased faster response times and transparency [1]. Legacy players are now compelled to compete with "insurance techs" (Insurtechs [2]) who are coming up with more useful and innovative approaches to engage with the customer and to help them achieve their objectives in a faster and economical way.

According to i2S experience, this kind of innovation carries its challenges in:

- Information processing, privacy, and protection regulations;
- Information Technology security;

- Digital identity authentication regulations;
- Business model regulations.

Insurance companies can now leverage technologies to operate at high speed and in a reliable way. Thus, the first step to successfully adopt any new technology would be to understand the impact across the value chain and invest in the right technology.

The main idea behind micro frontends is to think about a web app as a composition of pieces that are maintained by autonomous teams that operate in distinct business domains [3]. Each team is cross-functional and develops its features from the database to the user interface. However, micro frontends is a recent concept, and therefore there are no guides for its use which require study to understand the best adoption approach, as the enterprise has made that decision. Initially only one application that targets insurance companies will be used to understand all the implications.

Related with the technologies, this work will be done in a team that develops a frontend using the framework Angular and the library React. The application that is going to be used for the study and demonstration has being developed by that single team, but in a near future other teams will contribute with components. The application needs to be prepared to integrate micro frontends from other teams. In terms of microservices integration, this thesis follows a path with the use of a single API Gateway which orchestrates the communication between the micro frontends created and the existent microservices.

1.2 Problem

Developers are used to creating a fully-featured browser application, that sits on top of a microservice architecture [4]. That application will over time grow and become more challenging to maintain, and that is why it is called a “Frontend Monolith” [5].

On the other side, exists the concept of micro frontends that according to Michael Geers [5] can be defined as a set of features that are maintained by independent teams with a distinguished area of business.

Thus, the problem is how to 1) allow a faster compilation time, 2) the use of several programming languages/several versions of the same programming language, and 3) use the work done by others and reuse components by adopting the concept of micro frontends with API Gateway in an application. The application under study will be used by many insurance companies around the world and will allow them to speed up their internal processes with new ways of executing existing processes presented in a user-friendly interface. The main reason to follow the micro frontends approach is to understand and take advantage of the new technologies, to separate concerns, to allow the scalability of the teams and to build a set of frontends that could be easily separated and sold individually considering the client needs and requirements.

In a company-based perspective the most significant advantage is that micro frontends increase ownership because applications are split up according to specific domains of the website and increase domain knowledge within the team. This could lead to problems when teams are independent of each other: often when a team faces an issue it is most likely that this same issue also appears to the other team, so communication between teams is essential.

That communication will not also increase the knowledge of each individual team, team member, but will be guaranteed the consistency of the micro frontends, which will lead to faster delivery of better applications to the clients.

1.3 Objectives

The primary goal of this thesis is to develop a prototype that must be compliant with all architectural requirements with several smaller components, and each one will be able to be reused and tested separately. There are two specific implementation goals, which are characterized using the SMART indicators [6]:

Objectives:

1. Propose a solution to replace a Frontend Monolith:
    a. S.M.A.R.T indicators:
        i. The number of micro frontends developed during the elaboration period: at least 2;

ii. Reduce compilation time. At the time as these indicators were defined, the current frontend was allocated in an AWS instance with four T2.large instance types with the following characteristics.

- vCPUs: 2
- Memory: 8 GB
- 3.0 GHz Intel Scalable Processor
- With this setup in the current build pipeline, the average compilation time target of 10 measurements is 5 minutes.

2. Allow the use of several programming languages/several versions of the same programming language:
    a. S.M.A.R.T indicators
        i. Until the end of this thesis, at least 2 frontends built with different languages/versions must work together in the same application.

1.4 Approach and Development Process

The work will start by analysing patterns and integration possibilities followed by other people. This analysis will allow a better understanding of the advantages and disadvantages of each pattern and integration.

To perform this analysis is conducted a case study that in this case, is a research methodology for software engineering researches since it studies contemporary phenomena in its natural context [7].This means that the researcher does not set up an environment where the factors can be controlled. By doing this, the researcher can understand how the phenomena interact with the context, following a set of phases to conduct the case study:

1. Case study design: define objectives and plan the case study.
2. Preparation for data collection: describe procedures and protocols for data collection.
3. Collecting evidence: execution with data collection on the studied case.
4. Analysis of collected data

5. Reporting

Most of the work will be done using the Angular framework due to a company requirement. For demonstration purposes a component will be developed in React to realize its independence from the framework. In Annex B is presented those mentioned programming technologies straightforwardly.

During this work, different approaches to implementation will be examined and compared with the consideration of all requirements for the application. Once that work is finished, it will be detailed an architectural design considering all drivers, patterns, and concepts identified.

Active Reviews for Intermediate Designs(ARID) [8] or Architecture Trade-off Analysis Method (ATAM) [9] may be used to evaluate technical trade-offs. In addition, to perform an assessment of technical issues, but also economic ones, architectural decisions and to make adjustments to the design it is going to be used the Cost-Benefit Analysis Method (CBAM) [10]. Those methods are referenced in Annex C.

1.5 Document Structure

This document is structured in several chapters:

- **Introduction:** is the initial presentation of micro frontends and contextualization of the problem. This chapter also describes the main objectives of the work and the approach to accomplish the project goal;
- **State of the Art:** presents a theoretical framework of the given problem;
- **Analysis and Design:** present TOPSIS evaluation following it by the domain model and the essential use cases with some detail to justify the decisions taken;
- **Solution:** present the implemented solutions and also tested solutions for the problem;
- **Evaluation:** present a metrics comparison between the initial solution and the final solution;
- **Conclusion:** presents the results taken with this thesis.

2 State of the Art

In this chapter, it will be presented the state of the art concerning the main topics of interest for this work: Integration of microservices and Micro frontends, including its history.

2.1 Integration of Microservices into the Frontend

With the increase in applications complexity, it is natural to have frontend applications, with specific needs and restrictions. So, the following sub-chapters explain a way of handling those requirements.

Applications based on microservices facilitate the independent development, deployment and enable the scalability of the systems. Those characteristics are the significant aspects of the microservice integrations in the development process of frontends applications (web, mobile and desktop) [11].

In general, microservices are an architectural style that consists of small and independent resources that are available through the network, using, for example, a REST-based interface.

Each microservice has its functionality and run as a standalone process.

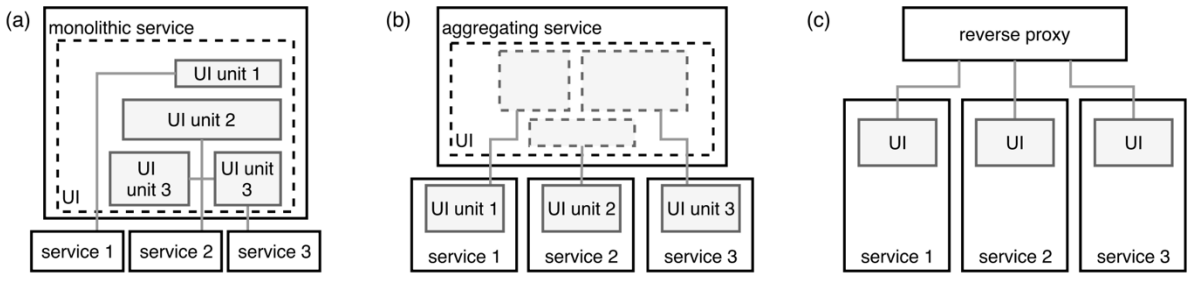

Figure 1 - Integration of Microservices
Source: [10]

Figure 1 illustrates relevant architectures and patterns that can be used for the integration of microservices in the frontend.

In this figure it is possible to observe three ways of performing this integration.

The first example represents a frontend application that is part of a unified application which communicates with its subordinated microservices via HTTP, interchanging data in a defined format, like JSON or XML [11].

Alternatively, and represented by the second example, microservices deliver complete fragments of the user interface that ultimately are gathered via composition using a higher-level application.

The last example represents self-contained systems. These systems subdivide an application along with defined technical barriers and each one is developed as a stand-alone web application that connects to other systems via hyperlinks.

In these forms of integration mentioned before, it is necessary communication between the services from the user interface and the microservices, so it is necessary the existence of a way that allows the developers to route this communication using a server to encapsulates the systems architecture. So, considering that, the patterns API Gateway and BFF appeared.

2.1.1 API Gateway Pattern

This pattern provides a single-entry point for all the web clients and it handles the requests in one of two ways, like it is demonstrated in Figure 2:

- Requests are proxied/routed to the appropriate service;
- Requests are fanning out to multiple services.

By doing so, this pattern can expose different API for each client [12].

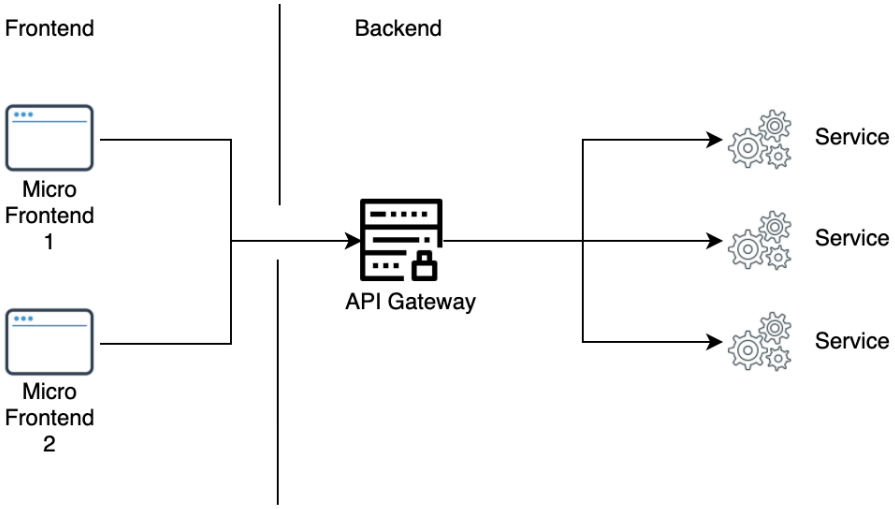

Figure 2 - API Gateway

2.1.2 Backend for Fronted Pattern

The Back-end for Front-end pattern [13] customizes back-end delivery for each user interface or experience, like it is demonstrated in Figure 3.

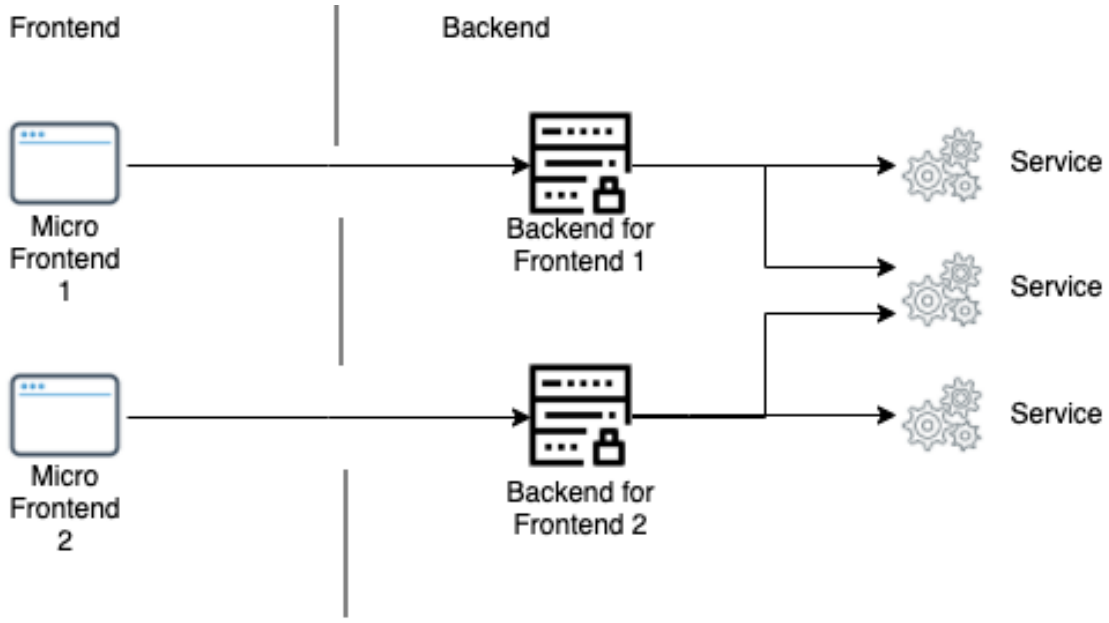

Figure 3 - Backend for Frontend

Before using BFF (Backend for Frontend) it should be considered that microservices must take into account only the customers specific behaviour and logic. Therefore,

business logic and other resources that are global should be managed elsewhere in the application.

Therefore, this pattern must be introduced by the team in alignment with each type of front-end, to ensure that each backend is adapted to the needs of each client.

2.2 Micro Frontend

Like microservices architecture, the micro frontends approach uses the same concept but on the browser side. This means that now it is possible to transform a monolithic web application into several smaller applications that run as one and each one has independent deployment and independent development [5].

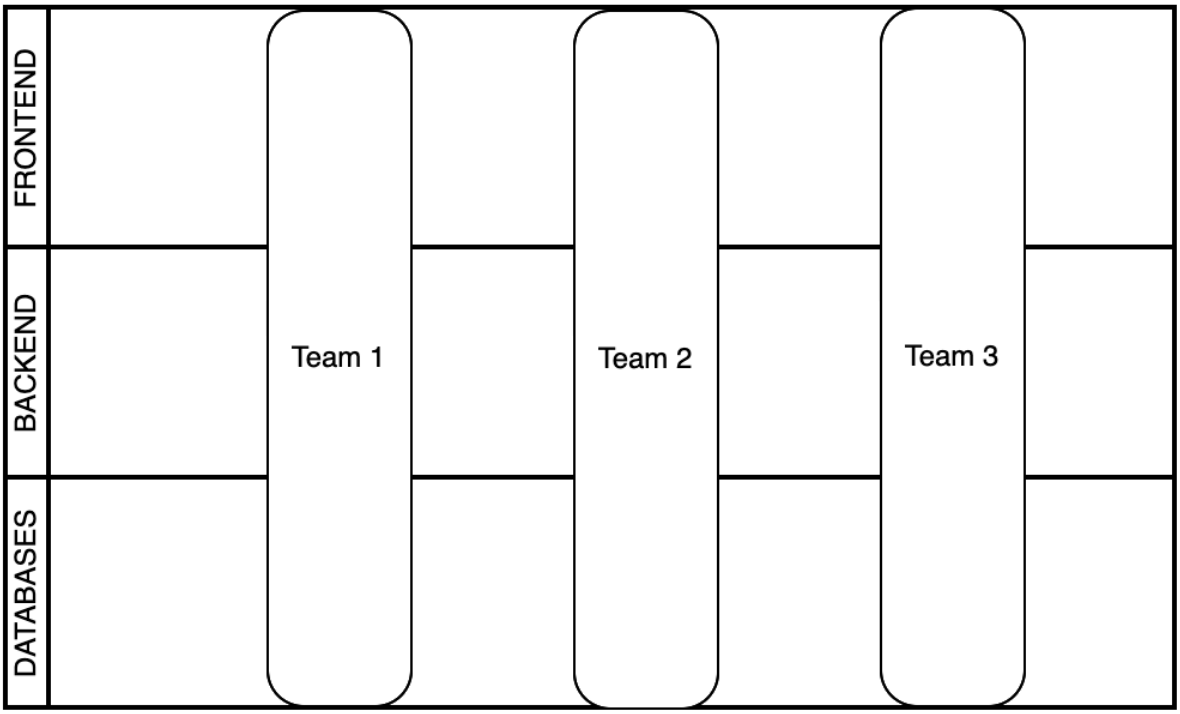

Figure 4 - Teams Separation in Micro Frontends

So, as demonstrated in Figure 4 this approach divides the application into vertical slices [14]. Each slice is run by a dedicated team, who built them from database to user interface (frontend). By doing that it is removed the need for a monolithic frontend and that brings benefits and downsides. The next sub-chapters will detail those benefits, advantages, and also the downsides.

2.2.1 History

In the current days, monolith frontend applications in big systems are becoming a reason for problems related to scalability, deployment, and especially maintenance. So, in November 2016, ThoughtWorks [15] recommended the appliance of the concept of microservices to the development of frontend applications, calling this approach micro-frontends. The idea behind this concept was to perform the separation of a single frontend application into a combination of several small applications satisfying some principles of microservices like being small, being focused on a single task and being autonomous [16].

2.2.2 Benefits

In this chapter, some possible benefits associated with a micro frontends approach, are detailed.

Incremental

Nowadays, the old and large monolith frontend is being held back by known technology or by code that was written under delivery pressure from deadlines. So, to avoid the disadvantages and the time spent on a rewrite, it is much more reliable and faster to separate traditional applications piece by piece and in the meantime maintain the delivery of new features. Once that idea is present, new frontend applications can be built considering that.

By doing so, developers will never go to be weighed down by old monolith frontend, and it will be guaranteed the incremental upgrades characteristic that this brings.

That doesn´t mean that in the future, no new features will need to be added in some cases this is the correct way, but now developers can make a choice based on the requirements. This is not the only thing good that comes from, micro frontends also introduce the ability to choose different frameworks to work with, based on the requirements [14].

Customer Focus

As said before, each team is responsible for performing all the life cycles of their micro frontend, so it means that each team ships their features and their development directly to the customer, which means that there isn´t the need to exist an operation team [17].

Source Code

By using micro frontends, the source code of each one will be consequently smaller than a simple monolithic frontend because each one will only have the dependencies that are needed. This will tend to a much more straightforward code for developers to work with and will decouple inappropriate dependencies between components that do not need to know about each other [14].

This is not a simple decision to make. Developers need to set their minds that they should produce code, taking into consideration all the good practices that all developers should know about. This means that from the beginning, developers should always pay attention to what they are producing clean and concrete code, where, for example, domain models are not shared between components, and data and events pass across distinct pieces of the application.

Cross-functional teams

One of the most significant advantages that are introduced by micro frontends is the structure of the team. That appends because people are grouped by different skills or technologies [14].

This brings at the front the concept of interdisciplinary teams, and there exists evident teamwork between frontends, backends, operations and businesspeople. And that comes with some upsides because in this case, people with different experiences and perspectives can come up with more creative and effective solutions for the development. It may not be the best solution in the world but, it´s right to assume that will be the solution that is more specialized in that team mission [18].

By arranging teams in a cross-functional way, it comes with the benefit of all members being directly involved in the feature. So, all the people get involved in that specific development making it easier for each of them self-identify with the product.

User interface

Each team generates the required source code (HTML, CSS, JavaScript) for a given feature and to facilitate their development use a specific library or framework to do that [14]. That means that each of them is free to choose the tools that better fit in their development and that they feel more comfortable to work with [19]. This means that teams can upgrade their dependencies without telling other teams.

Fragments

In most cases, frontend applications have some elements that appear on several pages, like menu options, for example, and it´s not a right approach that each team reimplemented that part of the code. And that is is when the concept of Fragments comes in.

A team can develop a part of the code (Fragment) to be reused by them or by other teams that don´t need to know about their implementation details. Each fragment must work without the dependencies of external components and may or not may need some contextualization like a product identification [14].

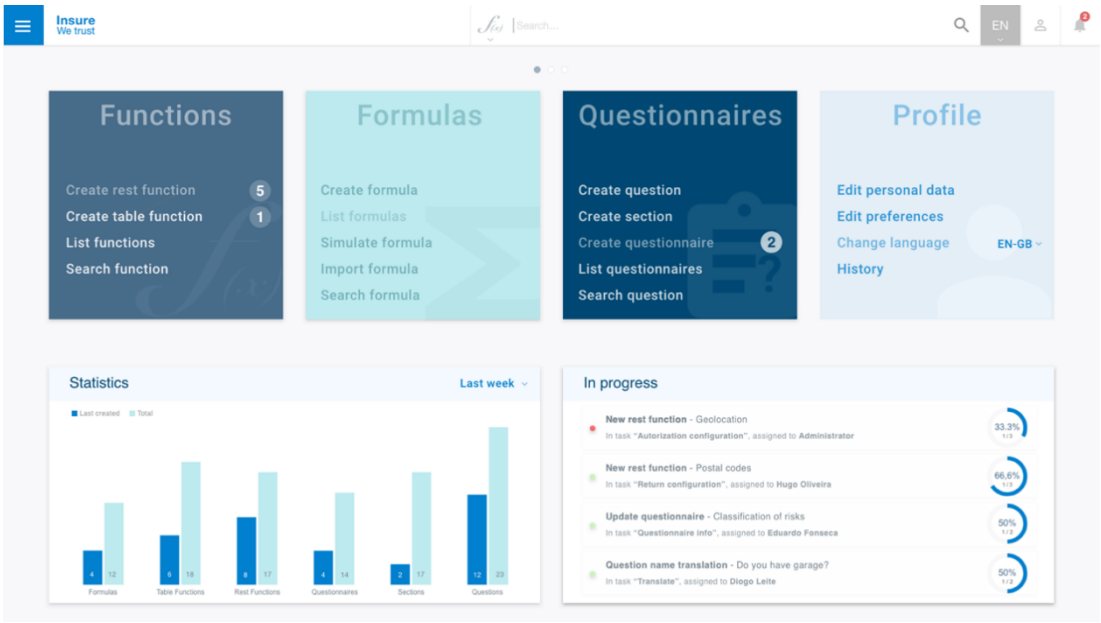

Figure 5 - Fragments

As an example, Figure 5 shows six blocks that could represent six fragments owned by different teams. Blocks are the boxes labelled as “Functions”, “Formulas”, “Questionnaires”, “Profile”, “Statistics and “In Progress”.

Transitions between pages

The transitions between pages can be done simply by clicking in an HTML link, which redirects to another route with different content. Taking advantage of the concept of single-page applications (client-side navigation), the rendering in the next page of the Fragments can be done without having to do a complete reload of them [14].

Deployment

In comparison to microservices, micro frontends have as key value its independent deployment. Each micro frontend should have its own build, like it is showed in Figure 6 should be tested independently, should be deployed alone, and teams should have the ability to push into production their code without being afraid of messing with other team code. That means that micro frontends should have a continuous delivery pipeline.

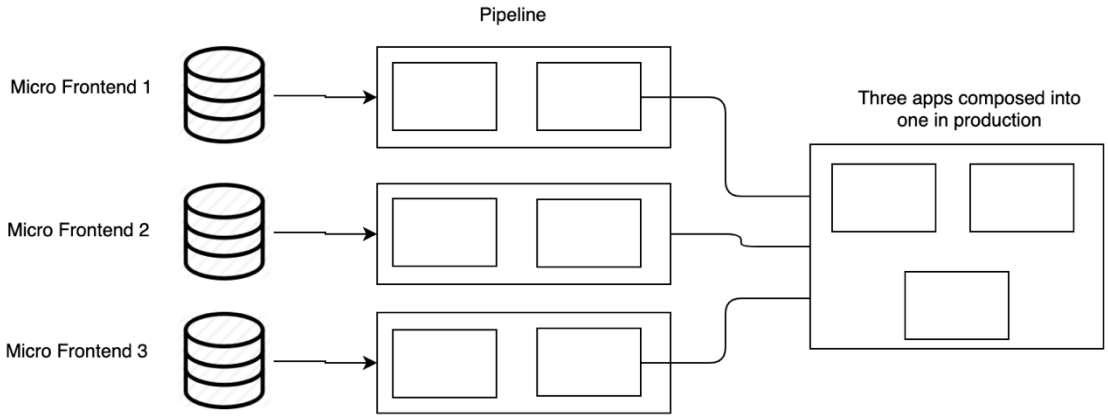

Figure 6 - Deployment Example

Like any architecture, micro frontends have their downsides. And that is what will be described below.

2.2.3 Downsides

In this chapter, some possible problems associated with a micro frontends approach, are detailed.

Payload size

As said before, following a micro fronted approach, each team can decide which dependencies each of them needs. This comes with a price, by doing so each part may have common dependencies between application components, which means that by splitting the monolithic frontend into five micro frontends and each one of them includes a copy of Angular, will force the customers to load Angular five times. This brings severe problems to application performance [3].

This is not an easy problem to solve. On one side exists the need to compile applications separately, but on the other hand, developers still want to develop them in such a way that they share general dependencies. One possible path to follow is to externalize those common dependencies. However, by doing this, in the future, it can lead to a big problem that is breaking changes in dependencies that consequently will need a coordinated upgrade. Taking into consideration all that has been said before, this is the crucial thing that must be avoided.

But not everything is bad news. Even if developers do not do anything and add all the dependencies that each micro frontend needs, it is possible that they still have better loading times because each page is loading its source code and its dependencies individually. Consequently, this leads to slower navigation between pages because on each navigation there is going to exists the need for reloading the source code and dependencies.

Environment

One of the values of building micro frontends is that teams should be able to build each fragment without the need of knowing the others. In some cases, teams may need to operate their micro frontend on a blank page instead of inside a container application. This brings advantages to the development process, especially if the goal is to migrate the old application to the new approach gradually. But some risks come along with this because the development environment could behave differently than the production one.

To fix this issue, teams may need to guarantee that people continuously integrate their micro frontend with environments conditions that are like production and test those environments to catch possible problems in integration [14].

Managing development

As more and more micro frontends are developed, will unavoidably lead to owning more repositories, more tools, more pipelines, more servers, and more domains do manage. So developers should always be taking into consideration two things [3].

- Is your organization maturity high enough to follow this approach without causing chaos?
- What should be a split to micro frontend and what should be a simple frontend component?

Consistency

This approach requires that each team has its database, but in some cases, one team needs some data from another team [14]. The most common solution to fix this issue is by doing data replication using, for example, an event bus or a feed system. But those replication mechanisms can consume a much time and may introduce latency, which can bring inconsistency data for a brief period. If everything is working as developers expected, this inconsistency can take a delay of some milliseconds or even seconds.

However, if by any chance something goes wrong, this duration can be longer.

Heterogeneity

As said before, one of the most significant advantages of micro frontends is the free technology choice, but this can bring some questions. People do not want those teams to have completely different technologies stack because, in the future, this makes it harder for developers to switch between teams.

So, every company should discuss the level of freedom in technology choice.

2.2.4 Applicability

After reviewing some benefits/downsides of micro frontends, it is time to analyse when do make sense to implement them.

Medium/Large projects

When using the approach of micro frontends and taking into consideration what was said before, teams should be split vertically [14].

Knowing that it is easy to understand that in smaller projects, it is harder to do that separation because the number of people is much less than in medium or large projects.

For example, if there are a 10 to 50 people split up into 2 to 6 teams, it is very likely that the vertical split up works very well. If otherwise, the number of people is at most 5, making a vertical split up will be more difficult. It wouldn´t make sense because of the scope of the project itself. In these cases, the concept of overengineering is applicable.

Browser compatibility

In some cases, exists some native applications that are monolithic by design, like iOS or Android. In these cases, there is the need to generate a single bundle that is submitted to

Apple or Google review process, and that is published, so replacing and composing a functionality on the fly is not possible [20].

To get around this issue, the loading of parts of that application from the web via WebView or via an embedded browser may be a possible solution. But doing this and implementing a native UI it is almost impossible to combine with multiple vertical teams.

One other possibility is that every single team have their web frontend that is exposed through a REST API. This API can now be used to add another user interface. Looking at this approach, it´s evident that the architecture sits on a horizontal monolithic layer that sits above a vertical team layer [14].

There are some downsides in this approach when targeting native apps, but it is still possible to implement micro frontends.

Faster development

One of the critical values of implementing micro frontends is to accelerate the developed process [14].

When following this approach, the amount of work that needs to be done is the same as implementing a monolithic frontend. However, the communication inside the team is faster and less formal because each team is focused on a single objective, and there isn´t the need of waiting for other teams and to discuss prioritization.

Monolithic Frontend

These days, most of the frontend applications don´t have the capacity of scaling, because in those projects only exists a single frontend that only one team can work with [14][21].

With micro frontends, that one single frontend is split into several vertical frontends, and if compared one with another, there is some benefits that show up.

Like,

- easier to test;
- smaller scope;
- smaller codebase;
- easier to refractor;
- independently deployment;
- easier to maintain and to understand;
- the facility of using the most convenient technology;
- isolates the risk of failures;
- is more predictable because it doesn´t share its state with other systems.

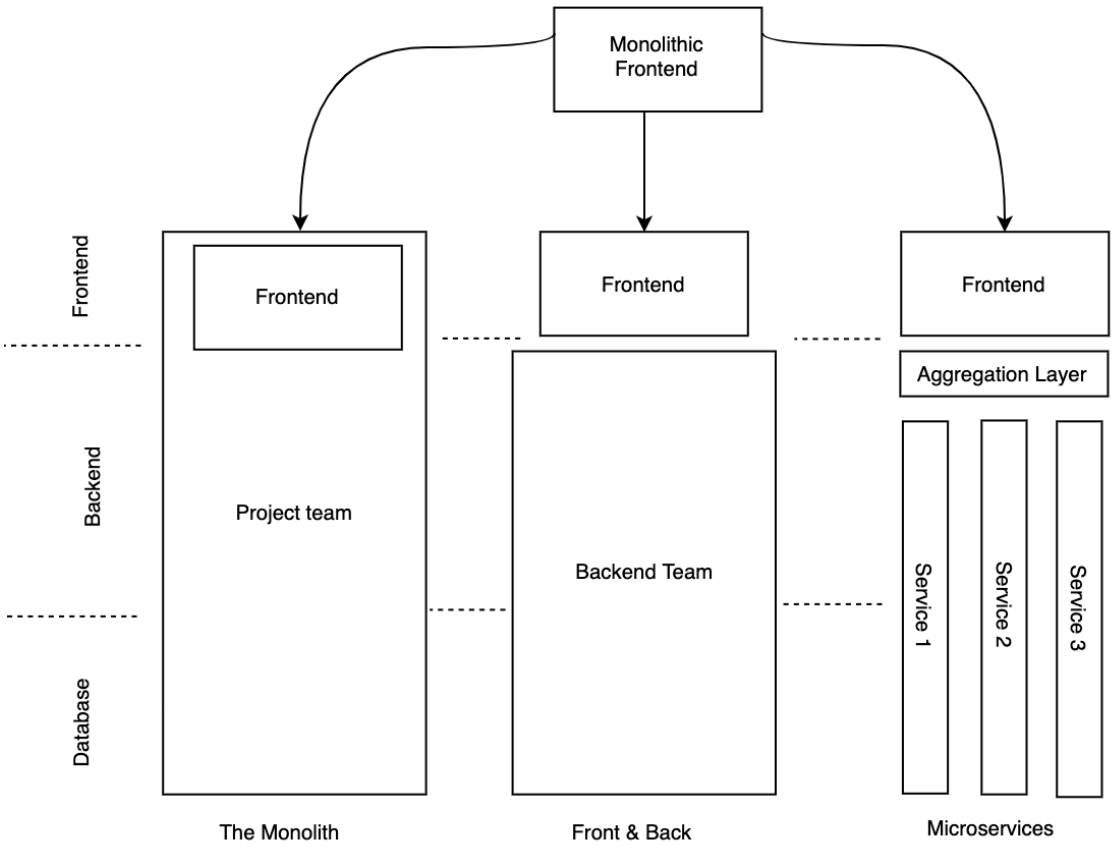

Figure 7 - Monolithic System

Figure 7 shows several representations of a monolithic system [21].

Follow changings

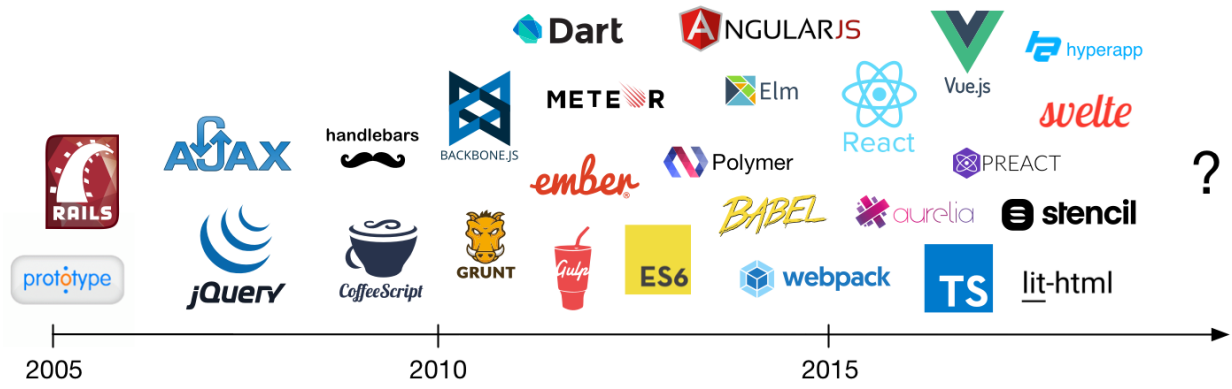

Figure 8 - Frontend History
Source: [22]

As shown in Figure 8, frontend technology is in constant change, so every developer has on its mind that the adoption of new technologies is part of the job [14].

But when the subject is a frontend, this is especially true, because frontend development is in nowadays a professional field and not anymore, a way of "making HTML prettier with some CSS." In these days, to deliver an excellent frontend to the client, the requirements like responsivity, usability, performance on the web, reusable components, testability, security and others need to take into consideration.

Legacy Systems

Refactoring legacy code and building migration strategies are in nowadays one of the things that take more time to the developer. Companies like GitHub, Trivago are two of the most significant players that recently are investing a considerable amount of time of work in this subject [14].

So, building an application and being able to refractor to new technologies in a more efficient way when they provide value to the companies is one of the fundamental values that must be taken into consideration. That does not mean that companies should always change their applications every time a new technology emerges [22].

Local decision

The ability to introduce and verify a new technology in a separate piece without having to interfere in other parts and without having to come up with some migration strategy is a valuable asset that micro frontends bring in a team level approach [14].

Since each team as full control of their technology decisions, it is easy for them to make changes without having to coordinate with others.

The only thing that must be taken into consideration is the conventions previously defined between teams, like namespaces or integration techniques at a frontend level [23].

Doing this kind of changes in a monolithic frontend would be considered difficult knowing that would require meetings to discuss the right approach to minimize the risks. But in a micro frontend approach, this would be particularly easy across time.

More Independence

Like it was said before, being autonomous is one of the fundamental values of micro frontends and microservices.

But even if across teams the technology stack is the same, and exists a homogenous environment, this approach also brings some advantages.

Self-sufficient

Pages and fragments bring their styles and scripts and should not have runtime dependencies.

This makes it possible for one team to perform a code update that may bring a version update to, without having to coordinate with other teams.

Knowing that each team brings their assets to the application in a homogeneous environment, might sound wasteful, but by doing this, teams can move faster and deliver features more quickly [14].

Shared nothing architecture

To enable the previously mentioned faster feature development, is critical that every component share dependencies as little as possible. Every single piece that is shared might create a management overhead [24].

Of course, some things must be shared, like fonts and styles.

Concluding, the core idea of micro frontends is reflected in the following aspects:

- Each team can complete the development independently;
- Each team can choose the framework that brings more benefits considering the development environment;
- Each team code is isolated, preventing shared runtime and global variables;
- Use prefixes to identify ownership of variables to avoid conflicts;
- Use browser events whenever possible to communicate.

2.2.5 Cross Concerns

One of the core ideas of micro frontends is that it should be developed by small and autonomous teams that have all the knowledge to build a feature that creates value for

the customer. But some concerns should be taking into consideration by all the teams together [14].

Performance

When the work of the previously mentioned small teams is assembled, the code generated can increase drastically. So, it´s crucial to always take into consideration the web performance.

Some things like, avoid redundant framework downloads or metrics and techniques that optimize the asset delivery, may help in this field.

Design

This may be the most critical concern that micro frontends should take into consideration. Every single team should build their frontend using the same stylings and design patterns so that when everything comes together, the final result has a unified look and feel for the customer [24].

Knowledge share

Being autonomous is very important in micro frontends, but it´s not productive when each team implements their HTTP interceptor. So, there must be times where exists a sharing knowledge so that teams can learn with each other [25].

And it is not the only problem. When it occurs adverse performance effects on the page, it´s not very easy to find the team responsible, so maybe there must exist an additional shared service to allow a better frontend integration. In the perfect world, this approach of boosting productivity should be higher than the complexity of managing this process [14].

2.3 Integration Frameworks

The following sub-chapters describe ways of integrating micro frontends.

Continuing with the concept of fragments, there must be a way of putting them in the right places. To do so, typically each team does not incorporate their content directly on the page. The correct process maybe passes by inserting a indicator at the mark-up where the fragment should go [14].

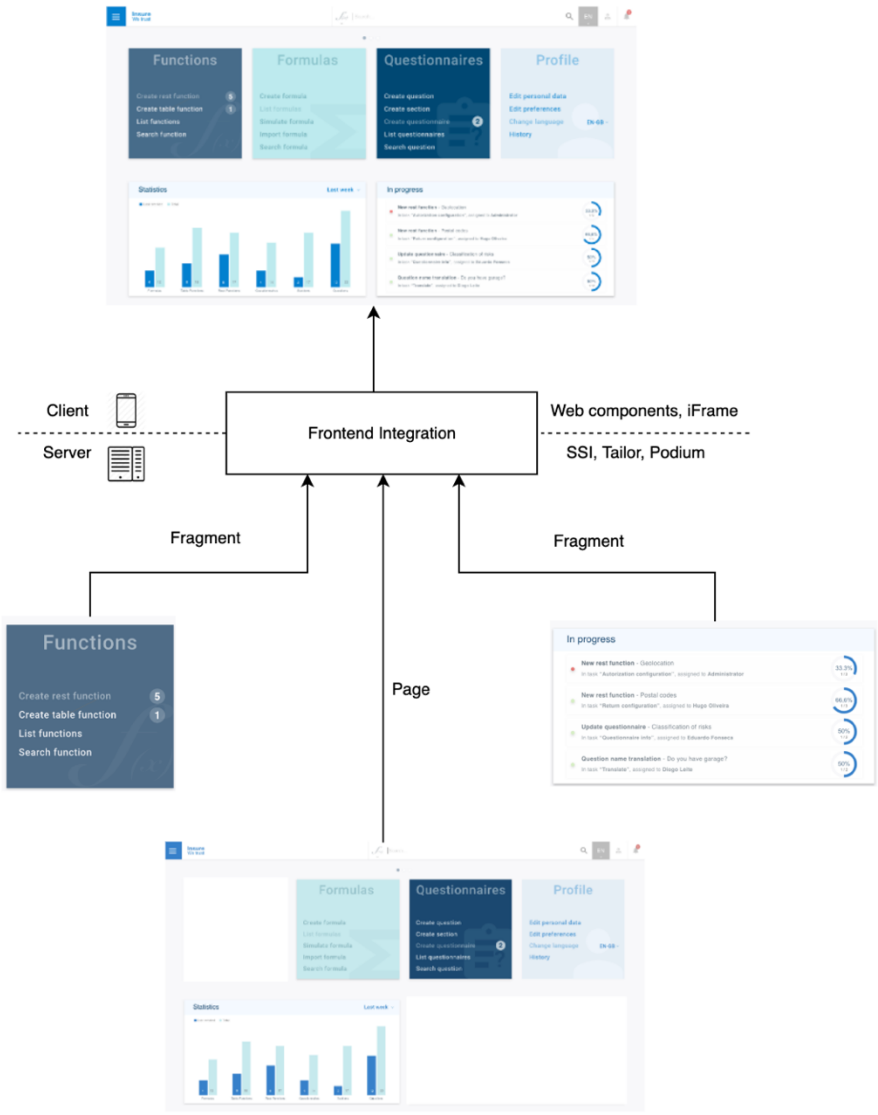

Figure 9 - Integration

This integration can be done in three different processes, and it is represented in Figure 9.

- Server-Side integration: SSI [26], ESI [27] or Podium [28];
- Client-side integration: Web components, Ajax, iFrames;
- Combination of both.

2.3.1 Server Side Includes

SSI is a server-side scripting language, more specifically, is a set of directives that are introduced in an HTML page and evaluated on the server while that page is being rendered.

This allows the dynamical addition of generated content into an existing HTML page. The critical value on SSI is that the dynamical addition referenced before is performed without the need of having to serve the entire page [29].

In conclusion, SSI is great when the target is small pieces of information that must be added to the web page.

2.3.2 Edge Side Includes

This server-side language provides a way for managing the web pages content dynamically across the application. That means that ESI, enable a means of developing web applications and at deployment time, where it should be assembled. With this, comes up a significantly reduced complexity, a shorter development time and smaller deployment costs [30].

2.3.3 Podium

Podium is a library that came up for building micro frontend and allows the page composition in a programmatically way without having to config or introduce a mark-up.

Each fragment produced using this library is isolated from the others, allowing him to be built in different technologies, to fail, to be processed and assembled without having to interfere with the other fragments [28].

2.3.4 Web Components

Web components is a web platform API that allows developers to create custom and reusable components that are used in other web pages with the encapsulation of HTML tags.

This approach is based on the web standards and features to support them are being added to the DOM specs, allowing web developers to extend HTML with encapsulated styling and custom behaviour easily [31].

2.3.5 AJAX

AJAX is a mix of client/ server-side development technologies which remove the need of a complete web page reload when a part of the application is updated. This means that the servers are less stress on the network and also allow faster operations, so, following this approach, the web pages become more responsive and more efficient [32].

2.3.6 iFrames

iFrames is used to display web pages inside web pages. It represents a rectangular area inside the DOM in which the browser will present a separate document. This approach also includes in its rectangular area scrollbars and borders [33].

3 Analysis and Design

In this chapter, it is presented a comparison between ways of doing a vertical decomposition for the frontend and ways of performing the backend communication to fulfil the micro frontends approach. Also, in Annex A, it is presented a value analysis of this project to understand its business area.

To perform this comparison and to choose the better approach to use, it is used the Technique for Order of Preference by Similarity to Ideal Solution method (TOPSIS). To do so, it is necessary a problem definition, structure a decision hierarchy and build a set of comparison matrix.

Following that, it is presented the architectural drivers associated with this thesis and the domain model of the project itself. After that, it is detailed the essential use cases to justify the decisions taken. Finally, a key pattern is discussed. It contributed to the choices made for the implementation.

This domain model only considers part of the application where this thesis sits on. So, the model presented in this chapter only represents the parts that need to be taken into consideration to fulfil the objectives described before in the chapter Introduction.

Insurance products are the templates from which insurance policies are issued. They have represented the structure and behaviour of the insurance policy, its formulas, wordings, rules, coverages and limits. They are complex because they are grounded in a complicated system and operating model.

In order to mitigate this complexity, it is necessary to understand the insurance product and the system behind it (people and IT). Only then it will be understood the inhibitors that slow down this IT area growth.

So, this thesis work is supported by a cloud-native software solution for insurance product development, that has the main purpose to enable insurers to faster develop insurance products, through configuration and automatic product generation. The cornerstone of this software is the product configuration and the capability of giving the end-user an understanding of how this configuration helps mitigate their insurance product related problems.

That configuration comes with two major forces that insurers should always be aware of: autonomy and efficiency. Knowing this and understanding the difficulties in the insurance area, the presented solution seeks on, among other factors, creating a set of reusable pieces, making the insurers work easier.

3.1 Possible Solutions

This section describes commons solutions used nowadays for micro frontends implementations [34].

3.1.1 Common Solutions

Route Distribution

Route distribution is the easiest-to-use approach. This solution looks like an aggregation of multiple frontend applications that are put together in a single one. In reality, those various applications are separated, which means that when a user wants to go from page A to page B its necessary a complete refresh of the page.

This routing distribution is often made using Nginx [35]. Code Block 1 shows an example of an Nginx configuration, and in there, it is possible to view that requests for different pages are distributed to different servers [34].

```
http {
  server {
    listen 80;
    server_name www.micro-frontend-example.com;
    location /api/ {
      proxy_pass http://http://173.31.25.15:8000/api;
    }
    location /web/users {
      proxy_pass http://173.31.25.29/web/users;
    }
    location /web/manage {
      proxy_pass http://173.31.25.27/web/manage;
    }
    location / {
      proxy_pass /;
    }
  }}}}
```

Code Block 1 - Nginx Configuration

iFrame

This approach creates a stand-alone hosting environment inside the HTML DOM.

To follow this approach, it is mandatory, taking into consideration that must exist a management application mechanism to know how to handle all the iFrames in the page and finally an application communication mechanism to define a set of communication specifications between all of the containers [34].

Application Microservices

Application micro-services means that each frontend is a stand-alone service-oriented application [34]. Each of them is built over unified application management and a start-up mechanism, for example, micro frontend framework Single-SPA [36].

Micro-Widget

This approach allows that different frontend application shares the same dependencies. So, it resolves the problem of repeatedly loading dependent files [34].

Micro-apps

Micro-apps solution is a consolidated integration, where exists a step-by-step splitting and recombination of the entire application in the process of building, pre-building and post-building.

This means that this approach satisfies three of essential characteristics of micro frontends: independent run, development and deploy [34].

So, in this approach, there are some common paths to follow in order to fulfil the intended requirements:

- Build each application/component independently, generating chunk files and in the end, build and categorize those chunk files. It is similar to microservices, but with higher costs;
- Develop each application/component independently and merge those apps in the integration process, generating a single application;
- At runtime, load the corresponding application code and template.

Figure 10 shows a representation of the paths mentioned before.

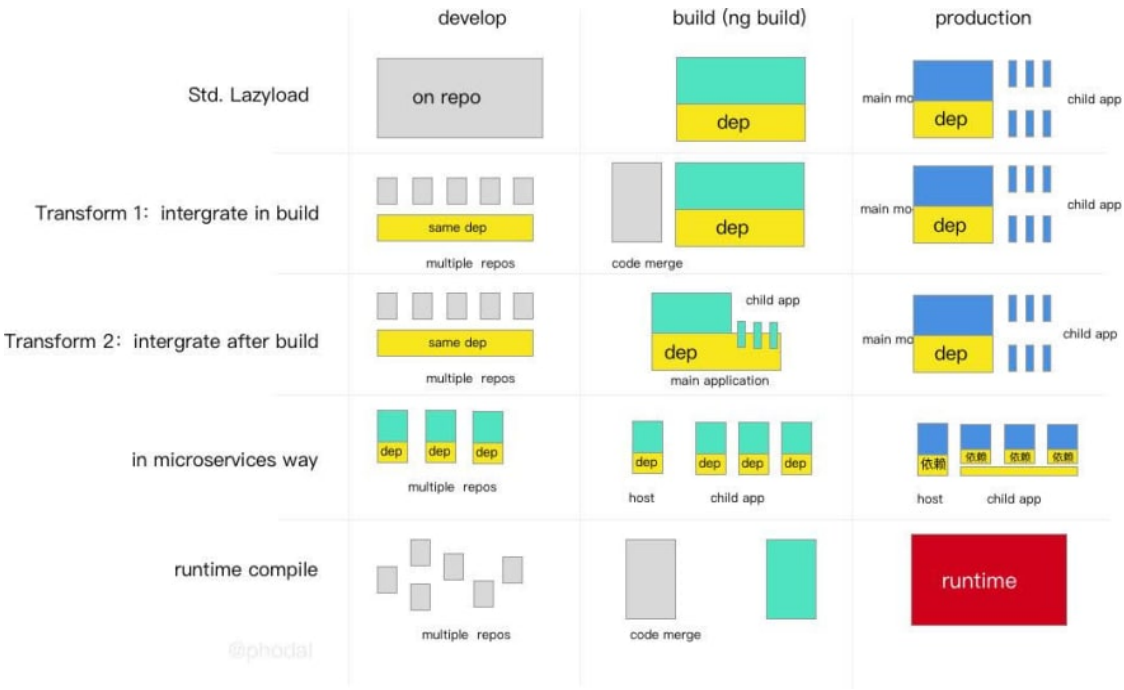

Figure 10 - Micro-app Paths
Source: [42]

Although this approach allows the development of multiple teams, it also comes with an inconvenient that is the requirement of all the applications/components must use the same framework.

The second inconvenient is that following this approach, the amount of specifications that it requires is enormous. That inconvenient means that teams must always specify the components and routes in order to avoid conflicts between different applications, construct complex scripts of building and have the same code in different applications.

Web Components

In the current days, exists two ways of building micro frontend applications with a Web components solution:

1. Integrating existing frameworks in Web components;
2. Integrate Web components into existing frameworks.

Starting with the first option, in nowadays frameworks have already the capability of supporting Web components. For example, Angular uses createCustomElement function to build a component in the form of a Web component. Code Block 2 shows an example of that integration [34].

```
platformBrowser()
  .bootstrapModuleFactory(MyCustomModuleNgFactory)
    .then(({injector}) => {
      const MyPCustomElement = createCustomElement(MyCustom, {injector});
      customElements.define('my-custom', MyPCustomElement);
});
```

Code Block 2 - Integrating Existing Frameworks in Web Components Example
Source: [42]

In alternately, developers can build their components straight into a component in the form of Web components and reuse them in any framework. That solution is represented in Code Block 3.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

import 'custom-components/customcomponents';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

Code Block 3 - Integrate Web Components into Existing Frameworks
Source: [42]

3.1.2 Comparison of the Possible Solutions

In Table 1 it is presented a comparison of the previously mentioned ways of implementing micro frontends.

That evaluation was based on several criteria that more fit to the company restrictions and that are more suitable to this thesis.

The criteria used were:

- Development costs;
- Maintenance costs;
- Independent deploy;
- Support different frameworks;
- Difficulties in implementation.

Table 1 - Comparison of the Possible Solutions

	Approaches					
Criteria	**Route Distribution**	**iFrame**	**Application Microservices**	**Micro-Widget**	**Micro-apps**	**Web Components**
Development costs	Low	Low	High	High	Medium	High
Maintenance costs	Low	Low	Low	Medium	Medium	Low
Independent deploy	Allow	Allow	Allow	Allow	Allow	Allow
Support different frameworks	Support	Support	Support	Not Support	Not Support	Support
Difficulties in implementation	Low	Medium	High	High	Medium	Medium

Looking at the data present in Table 1 is possible to verify some differences between the mentioned approaches. In a problem that exists the need for choosing the best approach, it is mandatory the criteria definition and a definition of some weight-associated to that each criterion.

So, to do that, the TOPSIS method is the one that best fits to perform that evaluation.

Technique for Order of Preference by Similarity to Ideal Solution method (TOPSIS) is an advanced technique that supports making decisions, quantifying intangible factors and evaluating choices in multicriteria decision situations.

In the first place, it was defined a decision matrix(Matrix M x N) where M = 6 approaches, N = 5 criteria, A_{ij} = Route Distribution, iFrame, Application Microservices, Micro-Widget, Micro-apps, Web Components and W_j = Development costs, Maintenance costs, Independent deploy, Support different frameworks and Difficulties in implementation.

For each W_j, it was defined weight, in which it was assumed, the following:

- Independent deploy (30%), Support different frameworks (30%);
- Development costs (15%), Difficulties in implementation (15%);
- Maintenance costs (10%).

Those weights were given taken into consideration this thesis objectives and also the seek of lower development costs and fast development that companies what to achieve.

Following the criteria, the definition, the next step passes by calculating the respective values.

Table 2 - TOPSIS Multicriteria Decision Matrix

Weights	**0,15**	**0,1**	**0,3**	**0,3**	**0,15**
	Development costs	**Maintenance costs**	**Independent deploy**	**Support different frameworks**	**Difficulties in implementation**
Route Distribution	2,5	2	1,6	2	2,5
iFrame	2,5	2	1,6	2	2,5
Application Microservices	1	2	1,6	2	0,5
Micro-Widget	1	1	1,6	1	0,5
Micro-apps	2	1	1,6	1	2
Web Components	1	2	1,6	2	2

- Calculate $(\sum x_{ij}^2)^{1/2}$ for each column;
- Divide each column by $(\sum x_{ij}^2)^{1/2}$ in order to obtain r_{ij};
-
    a. Multiply r_{ij} by the weight W_i to obtain the V_{ij};
    b. Determinate the ideal solution A^*.

Table 3 - Values for the Ideal Solution A^*

	Development costs	**Maintenance costs**	**Independent deploy**	**Support different frameworks**	**Difficulties in implementation**
Maximum value	0,085	0,047	0,122	0,141	0,083

-

a. Determinate the distance of the positive ideal solution A^*;

$$S_i^* = [\sum(V_j^* - V_{ij})^2]^{1/2}$$

Table 4 - Distance of the Positive Ideal Solution A^*

Si*
0
0
0,083
0,112
0,078
0,054

b. Determinate the distance of the negative ideal solution A^*;

$$S_i^* = [\sum(V_j^* - V_{ij})^2]^{1/2}$$

Table 5 - Distance of the Negative Ideal Solution A^*

Si'
0,221
0,221
0,191
0,144
0,166
0,201

- Calculate the relative proximity to the ideal solution.

$$C_i^* = S_i^- - /(S_i^* + S_i^-)$$

Table 6 - Relative Proximity to the Ideal Solution

Approach		
Route Distribution	1	Better
iFrame	1	
Application Microservices	0,698	
Micro-Widget	0,563	
Micro-apps	0,680	
Web Components	0,790	

Based on the metrics and the application of the TOPSIS method, it was determined that a route distribution or an iFrame approach are more advantageous and more adequate to the proposed criteria.

Thus, at the end of this evaluation, two approaches could be followed but exist some downsides in one of them that eliminate that option. As said in chapter Integration Frameworks, iFrames is an HTML document embedded into another website, and this brings three major disadvantages.

When using iFrames the HTML can be modified without the awareness of the websites creator, so malicious content can be attached without any permission. Another disadvantage is that in the current days, some browsers do not support them. Since the open and free nature of the internet is what draws users into checking out a website, this approach is eminently abandoned [37].

Considering what was said earlier, the best approach to follow would then be Route Distribution. However, this approach will be put aside in favour of the Web Components approach for several reasons.

The first reason is due to the fact that the project team on which this thesis seats on, is more familiar with this approach since in the past implementation tests have already been carried out that followed it.

The second and most important reason emerges from the possible need to integrate certain components in other applications of the company or in customer implementations. In other words, following a Web components approach, if this scenario happens, the integrating application would only need to include a javascript file where it considered necessary, thus having access to all the logic and features that such web component contains. Therefore, no implementation effort would be necessary on the part of the integrating application.

Taking into account what was previously mentioned, the next chapter describes two implementations of the same component in different frameworks implementations (Angular and React), following a Web Components approach.

3.1.3 Backend Communication Solutions

In this subchapter it is presented a visual comparison between API Gateway and the BFF pattern. This comparison serves to demonstrate a high-level view of the implementation carried out.

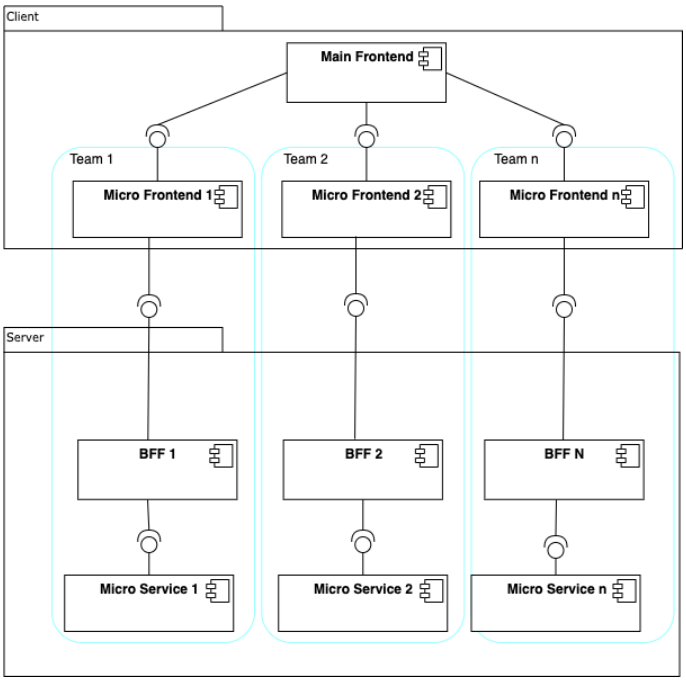

Figure 11 - BFF Pattern Solution

In Figure 11, it is possible to notice that each team has the responsibility to manage their microservice, their micro frontend and their gateway.

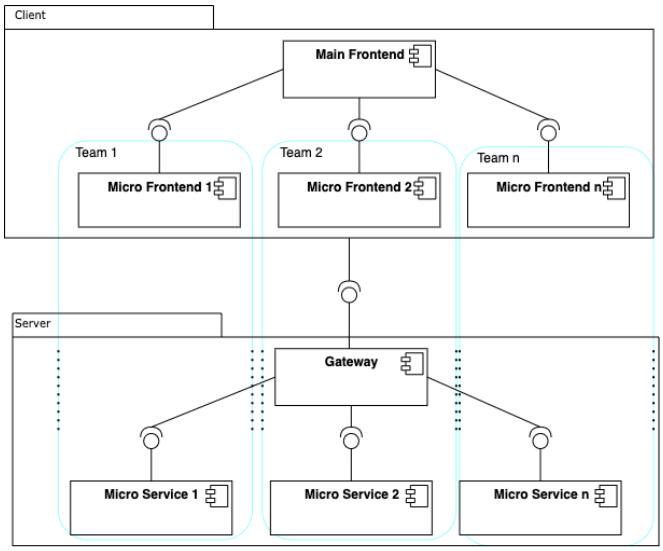

Figure 12 - API Gateway Solution

In Figure 12, it is represented the i2S solution for this project scope. In there each team is responsible for their micro frontend and for their microservice but exists a single gateway that is shared by all the teams.

3.2 Architectural Drivers

The following quality attributes for the system are derived from the business drivers that are detailed in subchapter ATAM Evaluation.

The software architecture will ensure achievement of system functions and the quality requirements of modifiability, openness, performance and availability as described in this specification.

Specific quality attributes for the developed software are as follows:

- Development process: a measure of time that takes to conclude the development process;
- Compilation Time: a measure of time in which the programming code is converted to the machine code
- Modifiability: the extent to which the micro frontends can be switched quickly and cost-effectively;
- Openness: the extent to which the micro frontends can be added and removed;
- Performance: a measure of the latency created for the loading and the usage of the micro frontends;
- Availability: the extent to which the micro frontends can be added and removed without a system failure.
- Compatibility: the extent to which the micro frontends can be rendered in most browsers

The Figure 13 represents a utility tree that help to concretize and prioritize those quality goals, considering the stakeholders opinions.

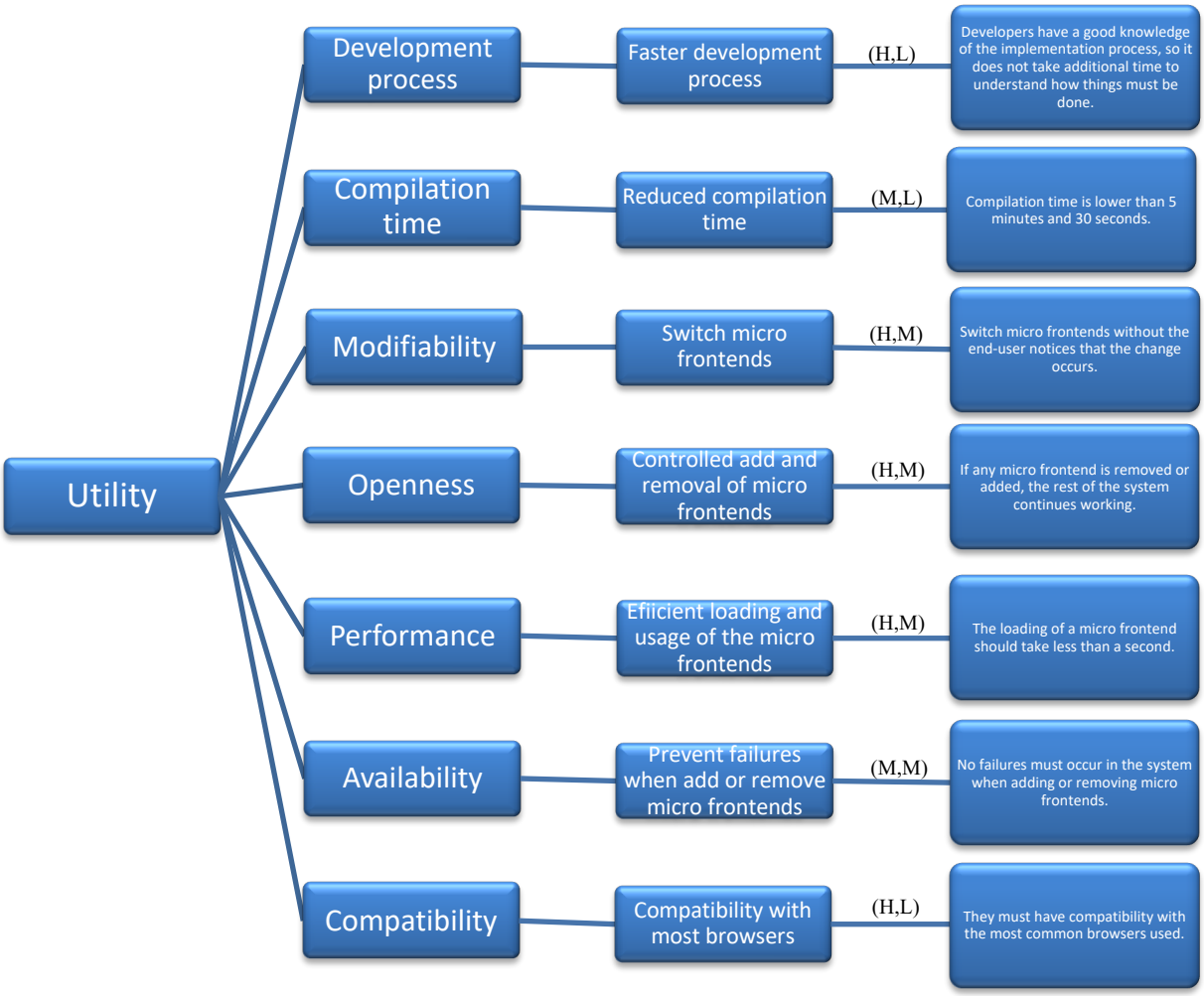

Figure 13 - Utility Tree

Those case scenarios are prioritized relative to each other. This prioritization is done using relative rankings like High (H), Medium (M) and Low (L) and it takes to dimensions.

The first dimension is the business importance and the second is the technical risk posed by the achievement of this scenario.

In addition to the scenarios described, there are characteristics inherent to a micro frontend approach that are reinforced here given the importance for the organization, namely:

- Each micro frontend is a perfectly isolated Web component and can have its style, its specific dependencies, and can use different frameworks and libraries from the

rest. So, for example, it should be possible to have one micro frontend in React and one in Angular, without any conflict.

- Each micro frontend has its own CI / CD pipeline independent of the others.

3.3 Domain Model

This subchapter shows a representation of the domain model (Figure 14) and a table (Table 7) that provides additional details of the entities presented.

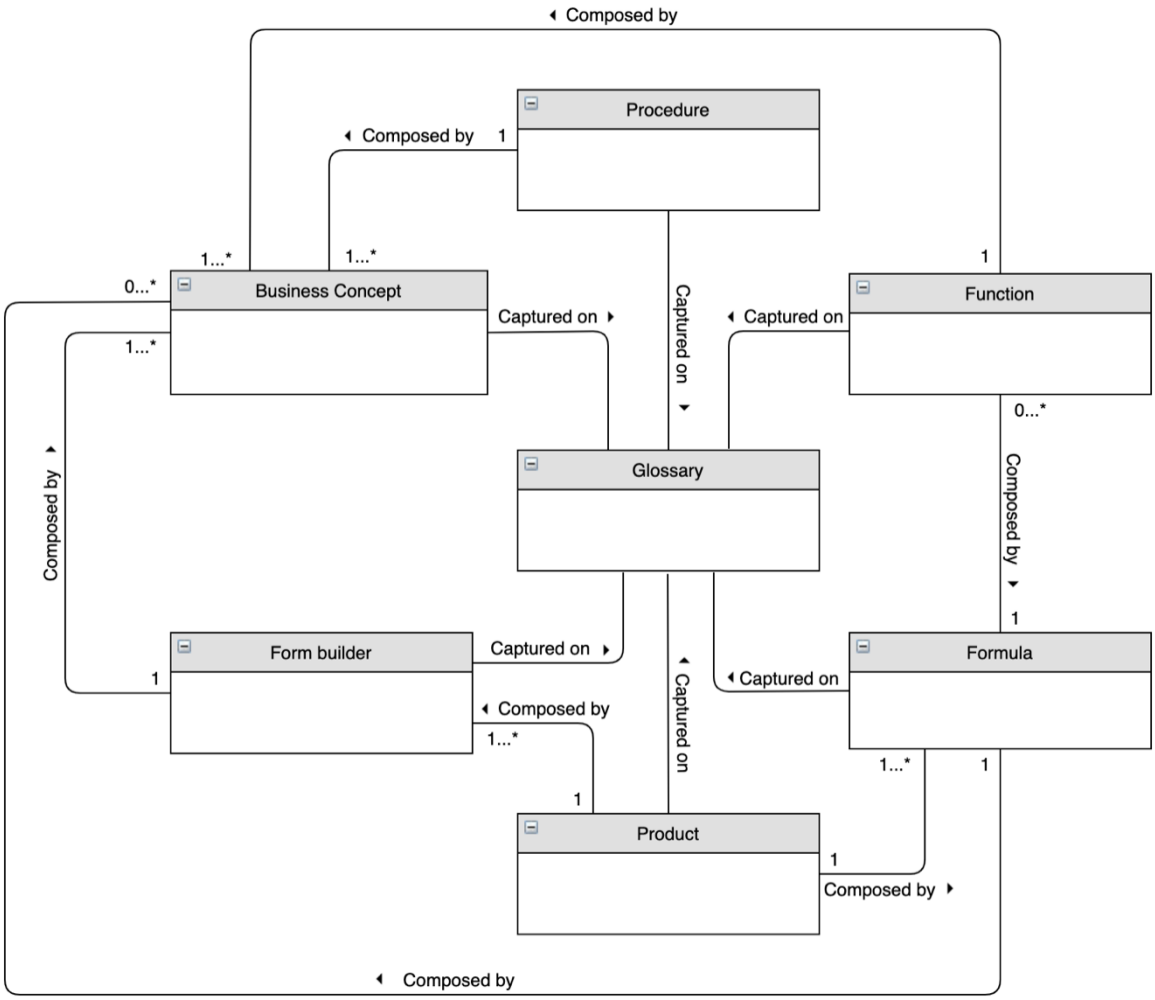

Figure 14 - Domain Model

Table 7 - Entities Descriptions

Entity	Description
Business Concept	Includes all the insurance concepts, which means that this entity includes everything that brings value in the insurance area.
Function	Use information from excel files or external API's to perform calculations based on the contained data.
Formula	Introduces a way of aggregating different functions or business concepts. This aggregation is useful to build insurance formulas to be used inside the Product entity.
Procedures	Represent the internal and external workflows within this project. The internal workflows are used to manage the actions that are directly related to the project concepts and external workflows are used to manage external actions inside the project.
Form builder	Represents the construction of forms, questionnaires and checklists that can be used in a task of the workflows, detailed in the entity Procedures, or can be used isolated.
Glossary	Aggregate useful information from the other entities, centralizing all the business terms used in the application.
Products	Contains all the information that is required to build insurance products. That information could be functions (tables, external services, databases), formulas, clauses, rules, covers, etc.

All of these entities mentioned before are built together to output the concept of insurance product configuration that considers the insurer needs and expectations related to the product change.

As said before, the insurance area is complex and this leads to a big problem, where an insurance term (part of an insurance product in its clauses) means different things to different stakeholders. Take for example, the term “party” – its definition varies in [38] and [39] and since insurance clauses are legally bound, rigour is of utmost importance. This complexity grows considering that a product also means different things depending on the IT system (policy administration system, a rating engine, a claims system, or a marketing and distribution platform). These characteristics accompanied by the legacy

systems of the insurance companies make the reuse of the products extremely difficult and sometimes it is better to launch a new one rather than rationalizing existing ones.

So, this domain model tries to respond to those difficulties, by creating a relationship between those several entities.

That relationship will allow the growth of the product configuration, removing the complexity of managing the change of insurance products and also by being the first step to create an application completely configurated, with its pieces well-separated, in terms of backend/frontend concerns.

3.4 Proposal

To understand better the implementation proposal, first, it is necessary to know how the solution before the introduction of the micro frontends as demonstrated by Figure 15.

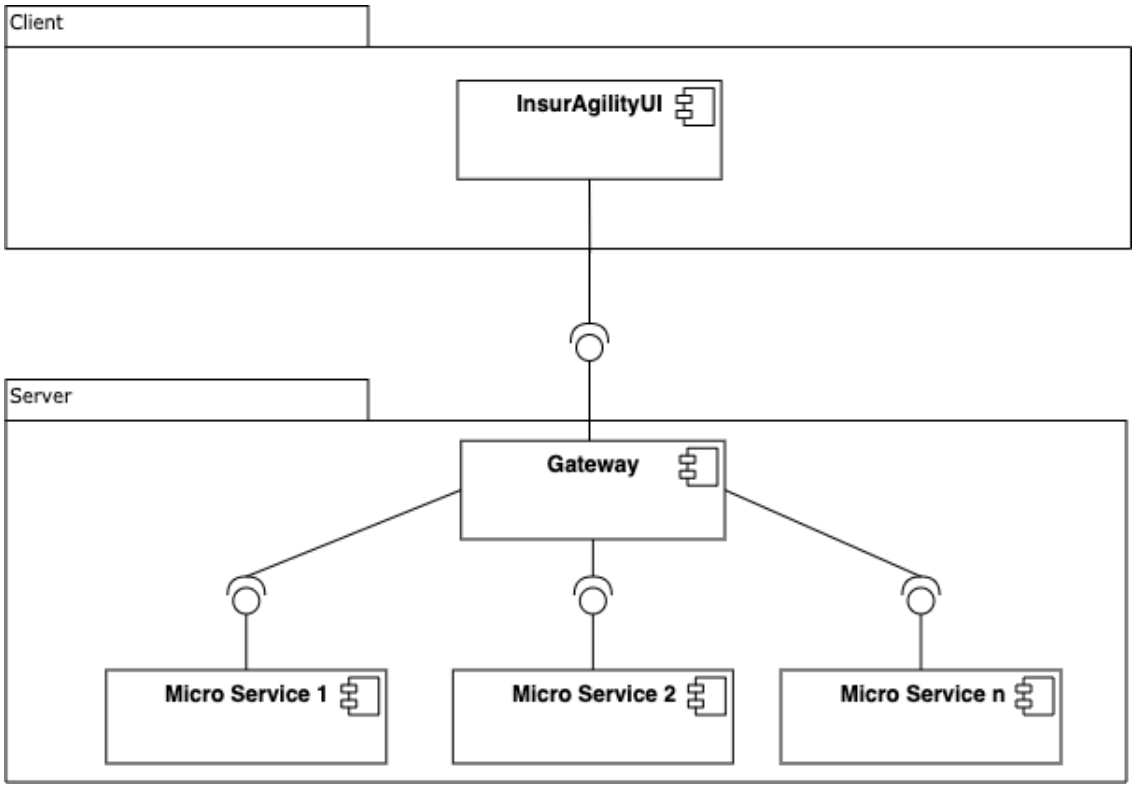

Figure 15 - InsurAgility Starting Implementation

In here, it is possible to verify the existence of a monolithic frontend that consumes several microservices.

Following the explanation given in the previous chapters, Figure 16 details how the implementation must work to fully complete the requirements.

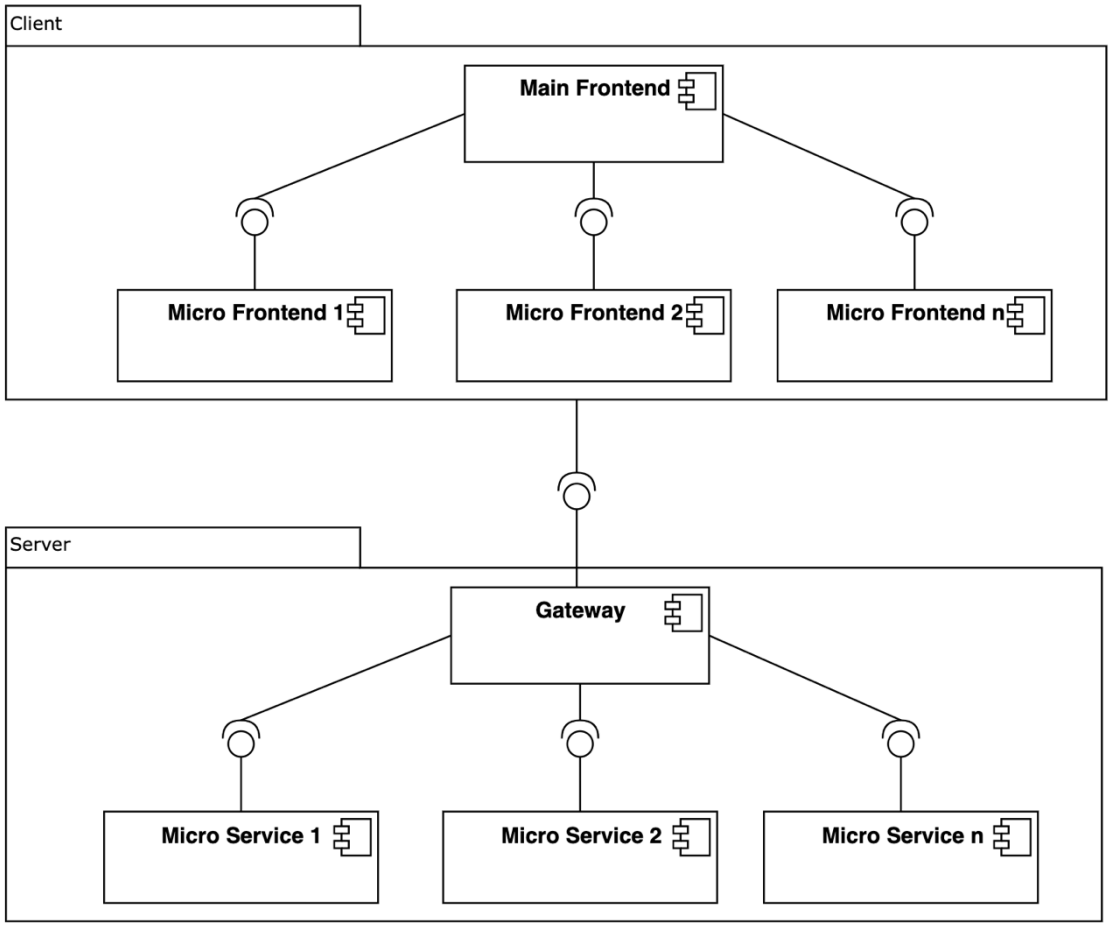

Figure 16 - General View Solution

So, in here it is possible to see that exists a main frontend that consumes the available micro frontends, by loading the JavaScript files generated when the micro frontend is build.

After that, each micro frontend, communicates with a gateway that orchestrates the requests and knows which micro service must be used.

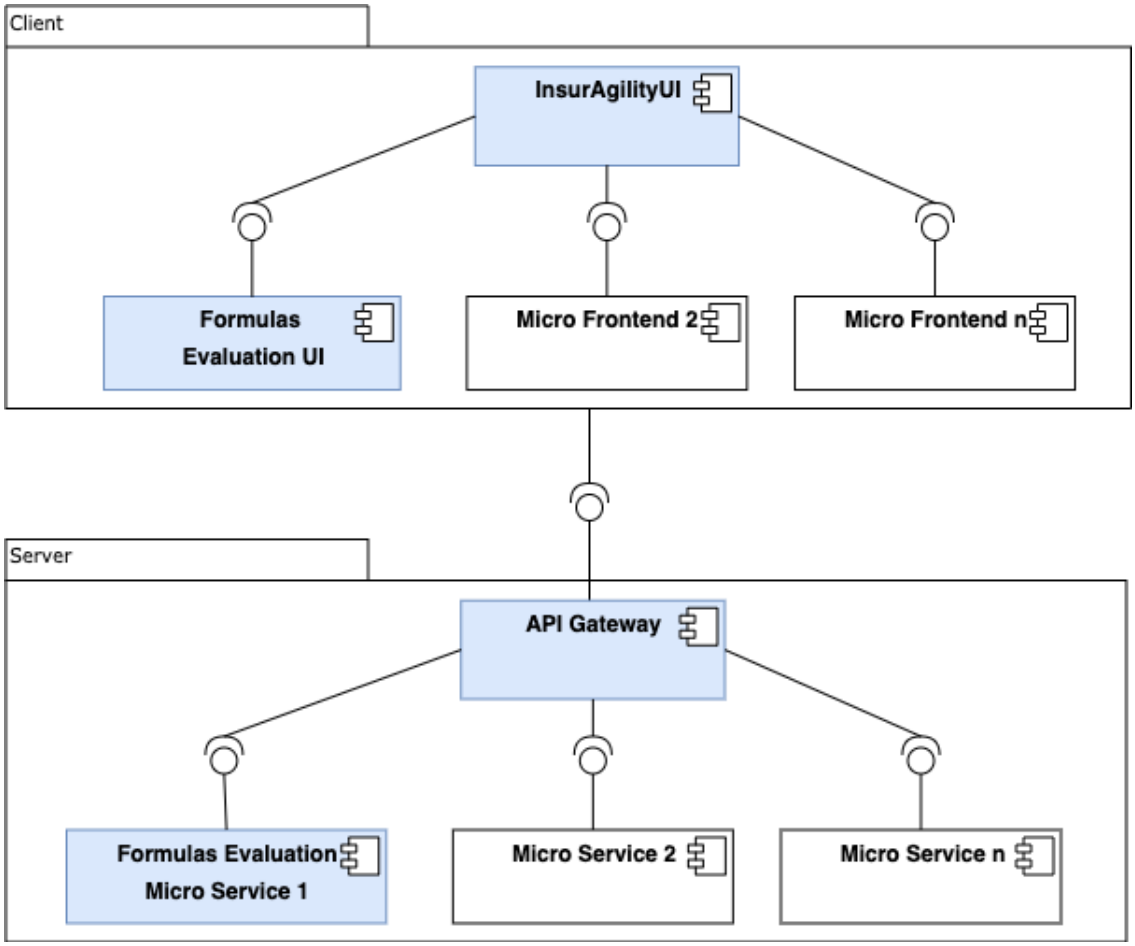

Figure 17 - Solution View

Figure 17 then represents the embodiment of the previous figure, where the components that are part of the solution are associated and defined by the blue boxes.

In this figure, an initial component that corresponds to the main application, InsurAgilityUI, is observed. This application then renders the user interface of the formulas evaluation. Then it communicates with the existing gateway who knows that it has to consume the micro service of the formulas.

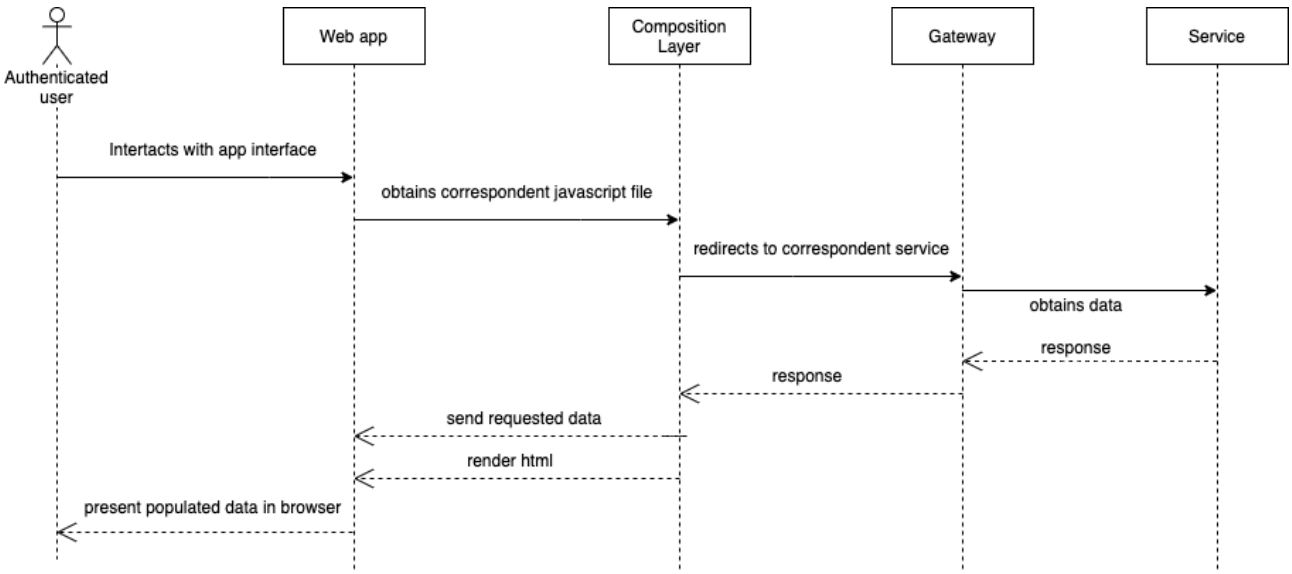

Figure 18 - Micro Frontend Sequence Diagram

Figure 18 shows a sequence diagram that represents the process of rendering micro frontends in a low detailed view. In here, it is visible a layer, Composition layer, that will treat the way as fragments picked and rendered. That layer contains all the necessary logic to know which piece needs to be rendered.

After that, the process passes by a gateway that orchestrates the requests to know which service that the fragment need.

In the end when the data and the JavaScript file that contains the micro frontend application are obtained, the populated data will appear to the user.

The logic behind of this use case that will serve to demonstrate this process described before will be related to Formulas Evaluations.

This consists of creating a set of business logic that takes the values present in a formula and performs mathematical calculations, Figure 19 illustrates this. Those calculations could arise through a simple simulation, that is, through the choice of certain values associated with the corresponding fields. But they could also arise from reading those same values through an excel file previously built by the user.

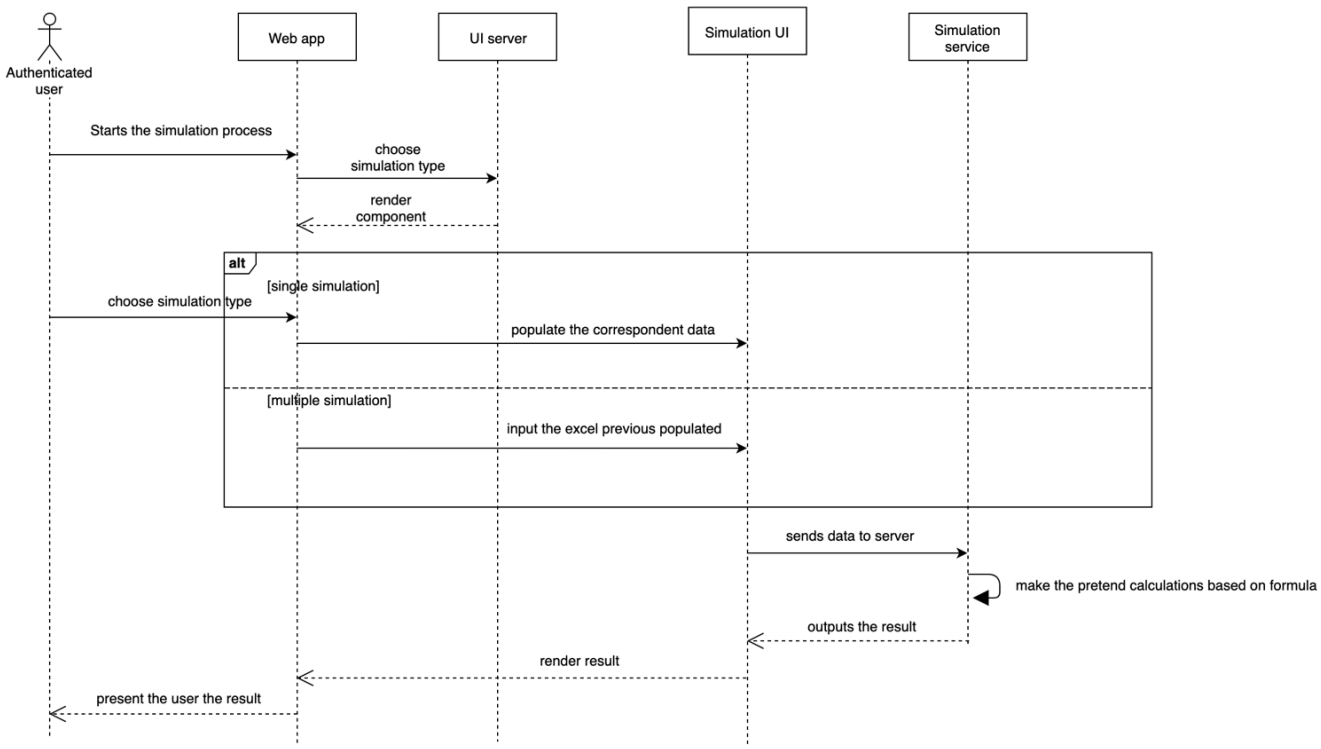

Figure 19 - Formula Simulation Sequence Diagram

The inherent complexity of this process is because the formula to be evaluated may contain other business entities associated with numerical values.

The decision to start with this concept to demonstrate the implementation of micro frontends with the use of a single API Gateway, fell on the conclusion that, at the moment, it does not make sense to separate the monolithic frontend that exists, taking into account reasons such as the fact that the frontend team is relatively small, 3 elements only, and that at the moment backend implementation is not fully ready to handler a micro frontend approach because it did not consider the BFF pattern.

For this thesis proposes and taking into account what was mentioned before, it is still possible to detail the frontend implementation necessary to follow a micro frontend approach.

Therefore, if this approach were followed, it would bring to it an enormous set of repositories, pipelines, servers and domains that would have to be managed by the team members. Consequently, this would bring development delays.

However, this implementation case, allowed to take a small piece of the application to demonstrate and document step by step the necessary way to build and integrate micro frontends in the specific context of i2S and the application in question.

Another reason that led to the choice of this implementation case is related to the ability of this component to be able to be isolated, since it only receives data from other components without any external dependencies. Finally, this choice was also because the team already knows the need that customers have in wanting to perform this type of calculations on their websites, without having to depend on the other entities mentioned before.

This will also allow the introduction of some shared concerns mentioned in the chapter Cross Concerns.

4 Solution

Before introducing this chapter, it is necessary to understand the steps related to the case study that was conducted.

As said in subchapter Approach and Development Process , to perform a case study, it is necessary to follow a set of steps:

1. Case study design: defined in chapter Introduction.
2. Preparation for data collection: described in chapter Introduction.
3. Collecting evidence: specified in chapter State of the Art.
4. Analysis of collected data: set in chapter Analysis and Design
5. Reporting: will be described in chapter Conclusion.

This chapter aims to specify the two implementations that were carried out and that emerged from the resultant analysis and design in the previous chapter.

Additionally, this chapter also describes two tested and discarded approaches implemented in an early stage of this thesis.

The logic behind this use case will serve to demonstrate the process described above and will be related to Formulas Evaluations.

In terms of the backend, both of the approaches use the same logic. Exists an API Gateway that orchestrates the HTTP requests made.

Figure 20 - API Gateway Configuration

Figure 20 represents the configuration that sets the routing and determinates for the endpoint invoked which should be the route called in backend side.

In this specific case, every time that the micro frontend calls the service "functions-unified-simulator", it will redirect to the functions microservice and perform the formula evaluation.

For security reasons, the “extra_config” field was omitted because in there was contained the encryption method that is used in this API.

4.1 Chosen Approaches

For this thesis, the chosen frameworks were Angular an React.

In terms of Angular the versions used was version 7, because at the beginning of this thesis the project context was sat on that version and 8 for the micro frontends because it was the latest released version at that same time.

Lastly, the React version used was 16 because like Angular version 8 was the latest released version at the time that the developing process started.

Relative to styling, the chosen approach of the CSS pre-processor was SASS, more specifically SCSS. This decision was made considering that encourages proper nesting of rules, encourages more modular code and allows writing better inline documentation [40].

4.1.1 Angular Implementation

The web component implemented using Angular followed the following steps.

First, a new angular project was created with the following command **ng new formula**, thus creating a new workspace project.

Install the following dependencies:

- ng add @angular/elements;
- npm i @webcomponents/custom-elements –save;
- npm i document-register-element –save;

- ng add ngx-build-plus.

Change angular.json:

- In the scripts include:

  {"input":"node_modules/document-register-element/build/document-register-element.js"}.

Change polyfills.ts:

- import '@webcomponents/custom-elements/src/ native-shim';
- import '@webcomponents/custom-elements/custom-elements.min';
- comment import 'zone.js/dist/zone'.

Then a new angular design was created within the previously created one. The reason why this happened, arise from the need for the existence of Web components that could be related to each other.

In this way it is then possible to create a web component that aggregates different parts of the application that do not make sense to be separated, thus allowing in the future the separation for micro frontends to be done in a way that will allow the clear separation of the teams and their responsibilities.

To allow this, the following command was executed inside the project directory created earlier: **ng generate application formula-evaluate**

Change app.module.ts of the newly created project:

- Remove bootstrap configuration;
- Import:
  - import { createCustomElement } from '@angular/elements';
  - import { AppComponent } from './app.component'.
- Add AppComponent in **declarations** and in **entryComponent;**
- Add inside AppModule the following Code Block 4;

```
constructor(private injector: Injector) { }

ngDoBootstrap() {
    const appElement = createCustomElement(AppComponent, { injector: this.injector });
    customElements.define('iaw-formula-evaluate', appElement);
}
```

Code Block 4 - AppModule Configuration

Change tsconfig.js:

- Change es2015 to es5;
- Add:
    - "resolveJsonModule": true;
    - "allowJs": true.

Change angular.json:

Change **outputHashing** parameter to "none" so that the generated bundles always have the same file names.

Add to scripts in package.json:

```
"pack:formula-evaluate": "cat ./dist/formula-evaluate/{polyfills,main}.js > ./../wc/formula/formula-evaluate.js",
"gen:formula-evaluate": "ng build --output-hashing=none --prod --project formula-evaluate --single-bundle && npm run pack:formula-evaluate",
"gen": "mkdir -p ./../wc/formula/ && npm run gen:formula-evaluate"
```

Code Block 5 - package.json Configuration

This step, represented in Code Block 5, is necessary to later generate a single bundled web component file.

After all these steps, all the necessary logic was added so that the objective described in the chapter Analysis and Design was implemented as demonstrated in Figure 21 and Figure 22.

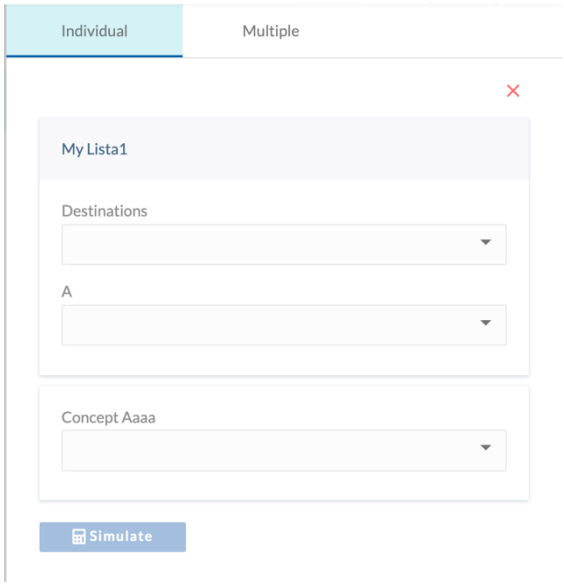

Figure 21 - Angular Micro Frontend, Individual Simulation

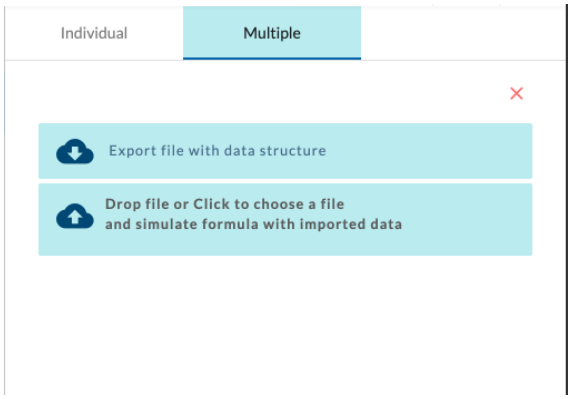

Figure 22 - Angular Micro Frontend, Multiple Simulation

4.1.2 React Implementation

Next, the necessary implementation in React is demonstrated in order to obtain the web component. This implementation is relatively simpler and faster than the one previously specified.

To perform this, it was used an npm package called **direflow-CLI** [41].

This package removes the extend boilerplate that comes with the developing process of building Web components in React.

The first step starts by installing direflow-CLI using the following command **npm i -g direflow-cli.**

After that installation is done, it is now possible to create a new Direflow setup using the command **direflow create** and choosing the right configurations, like name and language to be used (JavaScript or TypeScript). When this step is done, the only thing that is necessary to do is to add the business logic to fulfil the requirements need and run the command **npm run build** that will generate the web component resource file (.js).

Apart from that, the only thing that was changed in the infrastructure created by this npm package, was a configuration that sets the name of the file generated to be consumed as a web component.

This change occurred in file **direflow-webpack.js** and its represented in Code Block 6.

```
const { webpackConfig } = require('direflow-scripts');

/**
 * Webpack configuration for Direflow Component
 * Additional webpack plugins / overrides can be provided here
 */
module.exports = (config, env) => ({
  ...webpackConfig(config, env, { filename: 'evaluation-react.js' }),
});
```

Code Block 6 - direflow-webpack.js Configuration File

After all these steps and like in the Angular implementation, all the necessary logic was added so that the objective described in the chapter Analysis and Design was implemented as demonstrated in Figure 23 and Figure 24.

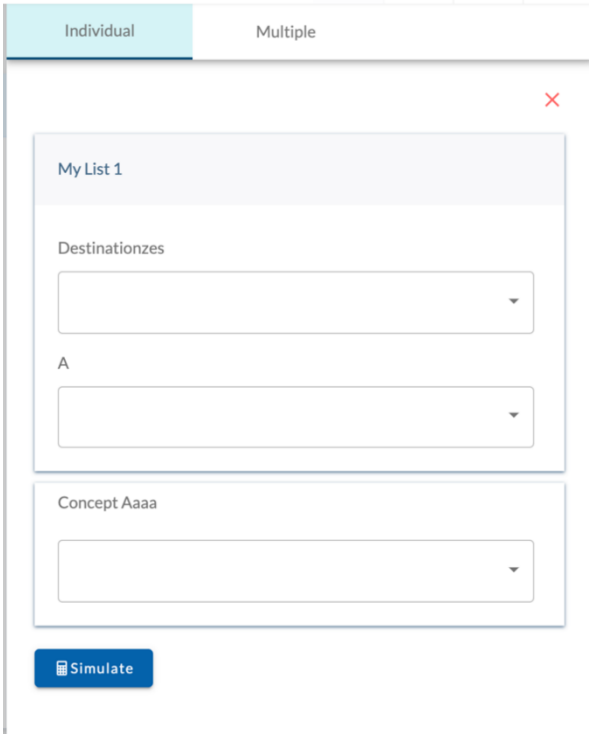

Figure 23 - React Micro Frontend, Individual Simulation

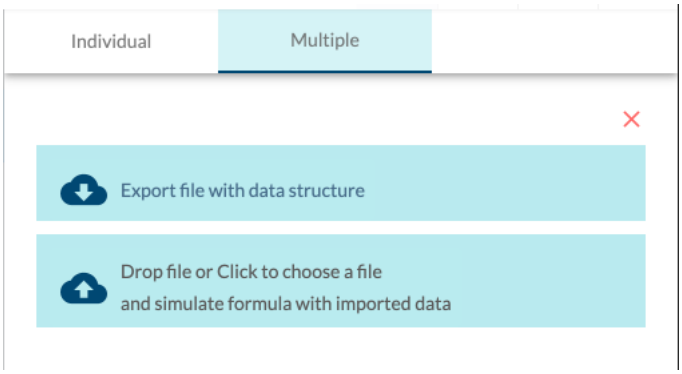

Figure 24 - React Micro Frontend, Multiple Simulation

4.1.3 Common Behaviour

The two implementations mentioned before, had the same objective at the business level as mentioned in subchapter Proposal.

Therefore, both had to enable the import and export of excel files, allow the choice of previously defined values, allow the management of notifications to notify the user of

the actions that were happening and also allow modal management to present the mathematical results.

Both implementations also had associated libraries that helped in the construction of the graphic components.

In the case of Angular, the Angular Material library was installed using the command **ng add @angular/material** and in the case of React, the Material-UI library was used through the command **npm install @material-ui/core**.

In terms of communication between web components and between the main page, there are some ways of doing that [42]. So, in this subchapter, it is detailed the ways of communication that were implemented.

The ways of pass data are the following:

1. Use a property: This consist of giving the value to the element, for example, component.data = myObj;
2. Use an attribute: To pass data into a component it is necessary to define which are the input attributes, that are coded in the component. After that it consists on writing the component selector like, <my-component data="myObj"></my-component>. When receiving the data passed is necessary to parse that in a JSON object to use the content.

The ways of passing data out are the following:

1. Events: Events should trigger when state changes in the component and when a user interaction occurs. When transferring data out with events it is possible to pass only relevant data in the event, reducing the amount of code that the components need.

Finally, and related to the availability of the web components, both implementations have the same behaviour. If by any case, the web components cannot be rendered, the main app remains functional and the space dedicated to rendering that component will remain white without anything on it.

4.1.4 Approaches Integration

This chapter described the way of integrating those micro frontends developed. In Figure 25 it is explained that behaviour, were its represented a white space that can render information prevenient from any micro frontend.

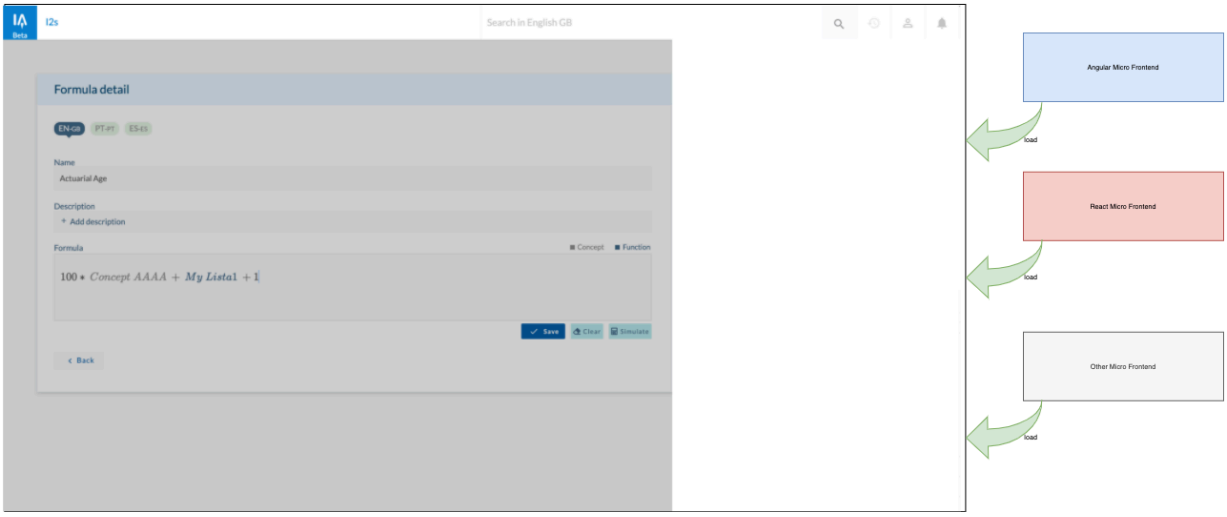

Figure 25 - Approaches Integration

As it was mentioned before, the chosen approach for implementation was web components and this consists of creating JavaScript files that are read dynamically when the application needs them. So, that is the only concern of the white space visible in the figure, the ability of loading external JavaScript files.

4.1.5 Backend Communication

As said by Martin Fowler[3] and related to micro frontends, there are some ways of structuring the relations between frontend and backend (API Gateway or BFF Pattern).

In the article mention before, exists a section called "Backend communication" that also refers that in some cases, it could not make sense to implement the BFF pattern depending on the project restrictions.

So, for this thesis, the chosen implementation will be based on the use of the API Gateway. And the reason why this path was chosen was based on several points that will be detailed up next.

1. The pattern BFF could be implemented for each web client (mobile or desktop) or for each micro frontend. In the case of this thesis reality, that only have a single web client (desktop) and looking at the following the excerpt “Are the clients owned by different teams? In this case, it might make sense to use the BFF pattern, because many times we do not want components with shared ownership in a microservice system. Each team could then have full ownership over its own BFF.”[43], it does not bring any advantage knowing that this project only has one web client.

2. In the case of this project scope, it is not wanted to have several public API's that could be easily acceded, and with the BFF pattern that will occur. In the implemented solution it is required to have a centralized access point that is delivered to the client through a set of endpoints with naming and controls defined by the team. So, this central point of access has the responsibility of retrieving data of internal services that our clients do not need to know that it exists, using the API Gateway this could be accomplished easily. For example, in the Formula frontend (previously mentioned) are consumed other objects from the business model that are managed by other services. So, in order to use those objects, the formula service invokes an endpoint called "functions-concepts" and the API Gateway is configured to redirect that endpoint to the concepts-service in order to obtain the correct data. In this way, it is guaranteed an abstraction between the name of the service that needs to be consumed and their implementations.

3. Another advantage of the use of API Gateway in the context of this project is the fact that it decreased the complexity of implementation, because there is no need of replication of the code, in terms of authentication and authorization, that are common concerns related to microservices. The fact of only exists a single mechanism of authentication helps to decide between API Gateway or BFF pattern. If there were several mechanisms the best implementation will be using the BFF pattern but in this case, this is not the reality [43].

For these reasons, the team follow an implementation using API Gateway.

4.2 Tested Approaches

In this subchapter, is detailed the early staged implementations that were tested and the reasons that lead to their exclusion.

All the approaches detailed bellow follow the Angular implementation that was described earlier. However, independently of the implementation choice, those approaches are ultimately excluded, regarding that in terms of business logic both do not bring any advantage and they were difficult to maintain.

Nonetheless, those approaches bring a tremendous volume of resources to be managed by the teams, like it was detailed in Downsides chapter.

Ultimately those web components, created in each of the following ways, not also come with a repetitive set of implementations that are common between each other but also come with an enormous amount of data that is created when the setup of each one is initiated.

4.2.1 Mapping one Micro Frontend to each CRUD Element

This approach, focus on creating a web component for each of the CRUD operations that any business object has (create, read, update, delete). In the context of this application the read, update and delete actions appear on the same screen, introducing the concept of inline-edit, like it is demonstrated in Figure 26. This means that the user has the capability of changing data, piece by piece, and deleting the object on the same page. So, this means that in terms of implementing, this approach requires two web components.

Figure 26 - Inline Edit Example

In this approach for each business entity, the number of web components needed increases to 2, like it is represented in Figure 27.

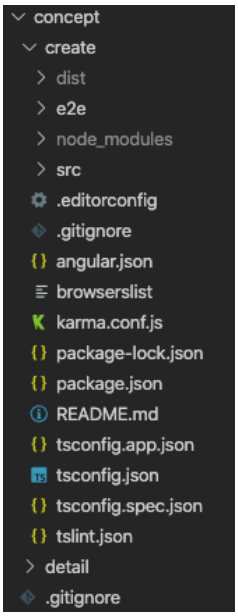

Figure 27 - One Micro Frontend to each CRUD Element

4.2.2 Mapping one Micro Frontend to one Business Entity

After the previous attempt of implementation, it was decided to try a new one, with a difference. This time the web component is created by each business entity. This means that each one has its own CRUD operations.

Comparing to the approach referred to earlier, the capability of inline-edit still exists.

In this way, and like its represented in Figure 28, the number of web components needed decreases by 50%, but some of those entities cannot be separated in this way from others. All because the business logic requires that, so ignoring this requirement brings a huge complexity in data events across all parts of the application.

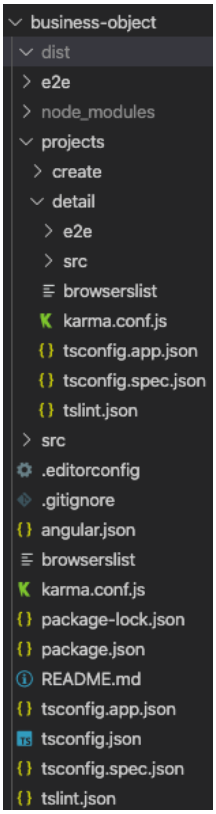

Figure 28 - One Micro Frontend to one Business Entity

5 Evaluation

This chapter presents an overview of the ATAM evaluation related to the designed architecture and the prototype implemented.

A critical analysis of the architectural decisions is carried out based on the ATAM method. This kind of method exposes the architectural risks that potentially inhibit the achievement of the organizations business goals based on the evaluation of the quality goals.

In this chapter it is also presented a comparison of the gathered metrics defined in the subchapter Objectives.

5.1 ATAM Evaluation

The Architecture Trade-off Analysis Method [9] (ATAM) is an analysis method that allows an architecture evaluation and it consists of four phases.

For this thesis, it is considered a lightweight version of the ATAM evaluation who is called Lightweight ATAM [44], which is supposed to be used internally by the development team without the need for the intervention from other stakeholders.

Also, this version of the evaluation did not require results from complicated evaluation procedures.

Presentation

The first phase is Presentation, detailed in the first chapter of this thesis, the Introduction, where it is given an enlightenment of the business side and it is also given a presentation of the target architecture, the micro frontends.

Major stakeholders are:

- Developers: the ones who developed the micro frontends;
- Testers: the ones who test the implemented solution and guarantee that it is in accordance with the customer requirements.;
- Customers: end-users, i2S clients, who are going to use the micro frontends.

Also, in this phase are detailed the business drivers that are motivating the development process and effort. Those business drivers are:

- time to deploy;
- ability to accommodate operational upgrades;
- ability to integrate new functionalities;
- ability to remove functionalities without failure;
- reduced development costs;
- separate concerns.

Investigation and Analysis

The second phase is the Investigation and Analysis, which is detailed in chapter State of the Art. It consists of explaining the general approach by presenting the quality attributes of the micro frontends approach, that is the context of this thesis.

Also, in this phase, it is identified the possible architectural approaches to understand how the proposed architecture supports each quality attribute. This analysis is made in chapter Analysis and Design and additionally includes the identification of the risks, sensitivity points and trade-offs of each one.

Finally, in this phase is represented the utility tree that illustrates the quality attributes derived from the business drivers. This utility tree is represented in chapter Architectural Drivers.

After determining the quality attributes, it is necessary to identify the trade-off points to which multiple attributes are sensitive that in this case is the current team size.

In this case, the team size was relatively small, only three people, and that was the reason why the author of this thesis decided that for the moment it did not make sense to make a full implementation of the micro frontends approach. Regardless of what was said earlier, this trade-off always influences teams that want to follow a micro frontend approach.

Knowing that it makes more sense to follow this approach with large teams, this trade-off positively influences, for example, the quality attributes like performance, compilation time, modifiability, openness, availability and compatibility in terms of

having more people to think in better ways of improving those qualities. But also influenced negatively the development process (slowing it down), considering that with the increase of the micro frontends, it also increases the number of servers and repositories to manage.

Testing

After that, comes the Testing phase. This phase is detailed in the chapters Analysis and Design and Solution, where are presented the tested and implemented scenarios.

Taking in what was made in the previously mentioned chapters, there was a brainstorm of the use case scenarios with the participation of all the stakeholders.

In this phase it was detailed the following set of use cases:

- build micro frontends for each CRUD operation;
- build micro frontends for each entity;
- build micro frontend for formula evaluation.

This last scenario was introduced considering all the limitations mentioned in the previous chapters and in conclusion, was the scenario chosen to continue with the micro frontends approach.

There was not any need for a voting process, that is characteristic of this phase because this step is omitted when using the lightweight ATAM.

Reporting

The last phase is Reporting. In this phase, a presentation of the evaluation results is given.

In this project, the methodology was used to detail the clear results in the evaluation of the build time resulting from the introduction of the micro frontends concept.

Taking into account the data collected and the implementation made, it is possible to observe that there were no significant improvements in terms of build time. The reason for this is inherent to the fact that at the moment this concept is not applicable in the application in question.

This methodology also made it possible to relate the best implementation approach taking into account the needs inherent to the application, such as the ability to reuse parts of the

application by customers, the ability to integrate the application in customer resources or resources of other applications of i2S.

In this case, the approach that best fits these restrictions is the Web Components approach, associated with the use case of formulas evaluation, since the team was more familiar with it and what has been studied is the approach that implies less development by the entities that want to use it.

Relatively to the quality attributes identified in chapter Architectural Drivers, Table 8 presents a review of them in the final prototype.

Table 8 - Quality Attribute Analysis

Quality Attribute	Verification in Prototype
Development process	The development process has increased because of two main reasons: • When a micro frontend is created, it is mandatory to follow a few steps(mention in chapter Chosen Approaches) before the development of the functionality starts. • When it is necessary to perform changes in the micro frontend, it is still required to generate the source file that will be consumed by the main frontend. So, to update a micro frontend takes more time than to update a component of the main frontend.
Compilation Time	This time is substantially short considering that the source code is separated into several projects, so there is lesser code to be converted, which means that the conversion time is faster.
Modifiability	Considering the development process, switching a micro frontend by another it is only necessary to change the generated file that is consumed and change the HTML tag(in the case that this tag change) that is associated to that micro frontend.
Openness	In the same way that switching micro frontends has an easy development process, the ability to remove or add a micro frontend has the same characteristic. To remove a micro frontend, developers only need to remove the HTML tag referred before and stop the loading of its generated file. In the case of adding, developers only need to add the HTML tag and start loading the generated file of that micro frontend.

Performance	Considering that the generated files of the micro frontends are not inside the main frontend, some latency may occur in the case of the internet connection is slower.
Availability	As said before, removing or adding a micro frontend has a fast and straightforward process. So, that period in which any of those actions are occurring, any kind of system failure will happen. But in the case of adding it may cause small period unavailability of that functionality or functionalities (depending on the micro frontend objective).
Compatibility	For some browsers, it may exist the need for developers to add some conditions or configurations to the implementation as said in chapter Cross Concerns so to render micro frontends, but that is not a common concern to be taken as browsers are following the technologies changes.

5.2 Comparison of Gathered Metrics

One of the aspects of this thesis was the evaluation of the benefits and harms that arise from the implementation of the micro frontends concept in an i2S application.

In order to understand the impacts that this approach brings, metrics were collected with the aim that in the end, it would be possible to make a comparison of the state of the application before and after the introduction of the micro frontend concept.

The project objectives and the metrics associated with each are presented below with the respective comparison of before and after.

The first objective is to propose a micro frontend approach and the values that will allow the measure are the following.

The second is to allow the use of several programming languages/several versions of the same programming language.

5.2.1 Propose a solution to replace a Frontend Monolith

To verify the work and assess if there is the need to do improvements, it was set two specific indicators that will allow that measurement.

The current frontend was deployed in an AWS instance with four T2.large instance types with the following characteristics.

- vCPUs: 2
- Memory: 8 GB
- 3.0 GHz Intel Scalable Processor

With this setup in the current build pipeline, the average compilation time target of 10 measurements is 5 minutes. The objective is to reduce this time.

The number of micro frontends developed during the elaboration period: at least 2

As demonstrated in the Solution chapter, during the development of the thesis, it was implemented four micro frontends, this number included the tested approaches that were later discarded, like it is demonstrated in Figure 29.

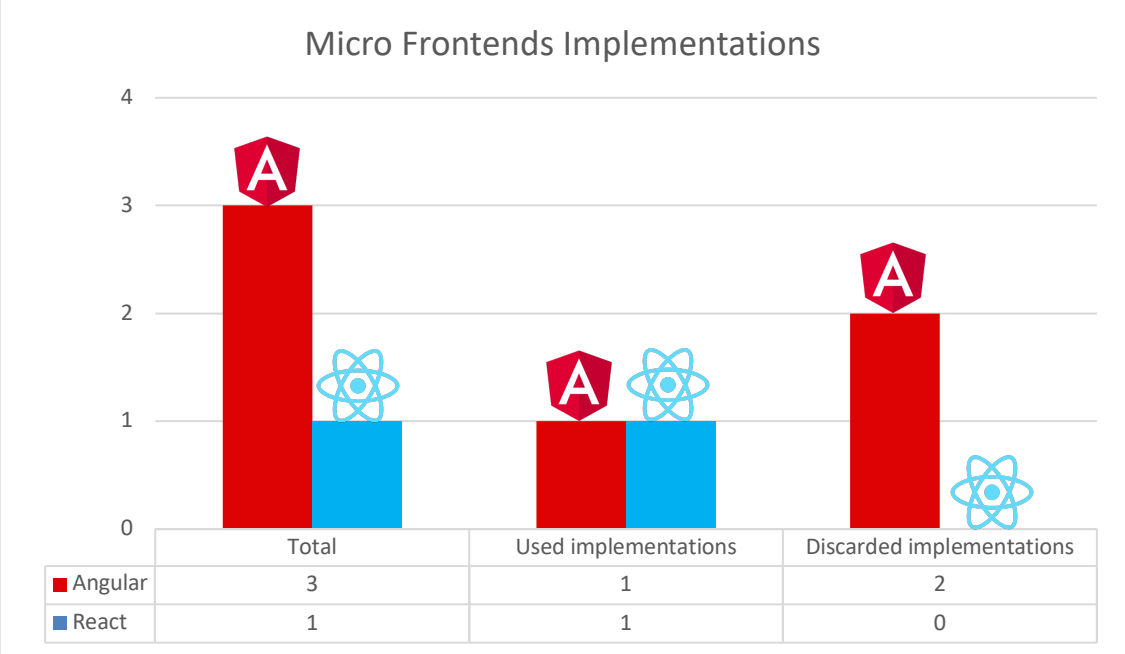

Figure 29 - Micro Frontends Implementation

It is possible to say that this particular indicator was indeed concluded with success.

Reduce compilation time

The evaluation methodology passes by doing an exploratory data analysis of the obtained results and also by analysing the result of 20 experiences, where the build time was measure. After that, it was performed a statistical analysis that verifies if the obtained results are meaningful.

These data were collected at the end of the thesis by the author.

Figure 30 and Figure 31 show the current pipeline before the introduction of the concept of micro frontends. In these figures, it is possible to see the current time that the context application has taken the last 10 times to perform the build step, on average is 5 minutes 8 seconds and 1 millisecond.

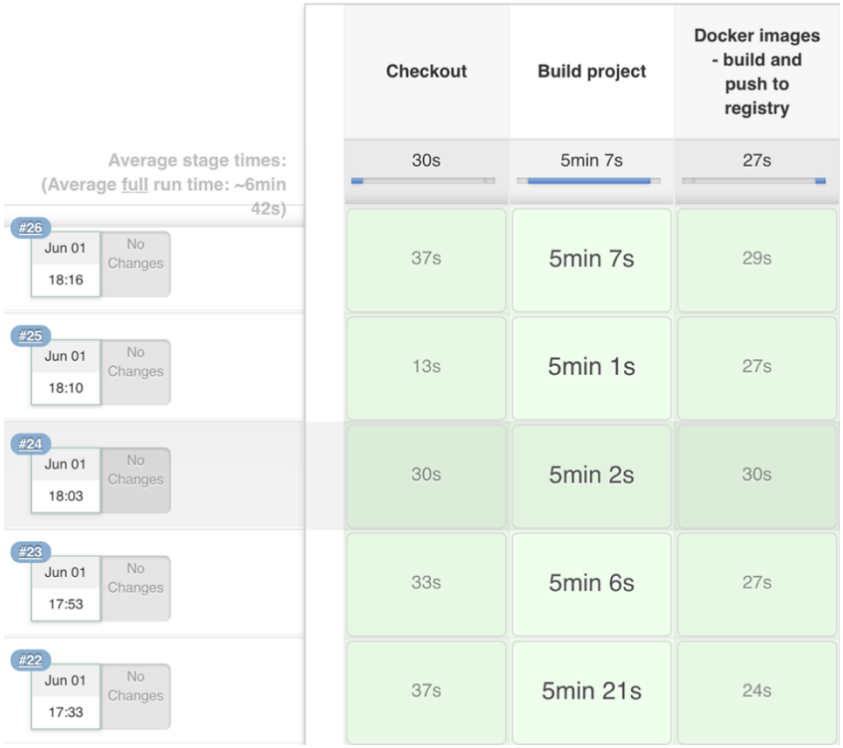

Figure 30 - Data Collected Before Micro Frontends

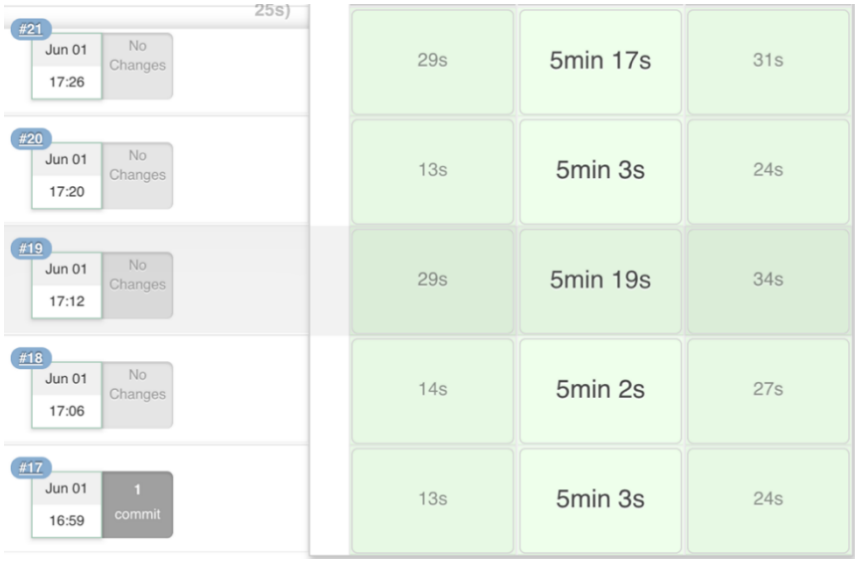

Figure 31 - Other data Collected Before Micro Frontends

After having these data, it was deleted the components that now are coded in the web components. By doing so, all the dependencies related to them were removed from the application and the build time was then measured again.

As it is possible to see, Figure 32 and Figure 33, also the current pipeline after the introduction of the concept of micro frontends, coded as web components.

Now, it is possible to see that in this way the build time of the last 10 times was 5 minutes 3 seconds and 3 milliseconds.

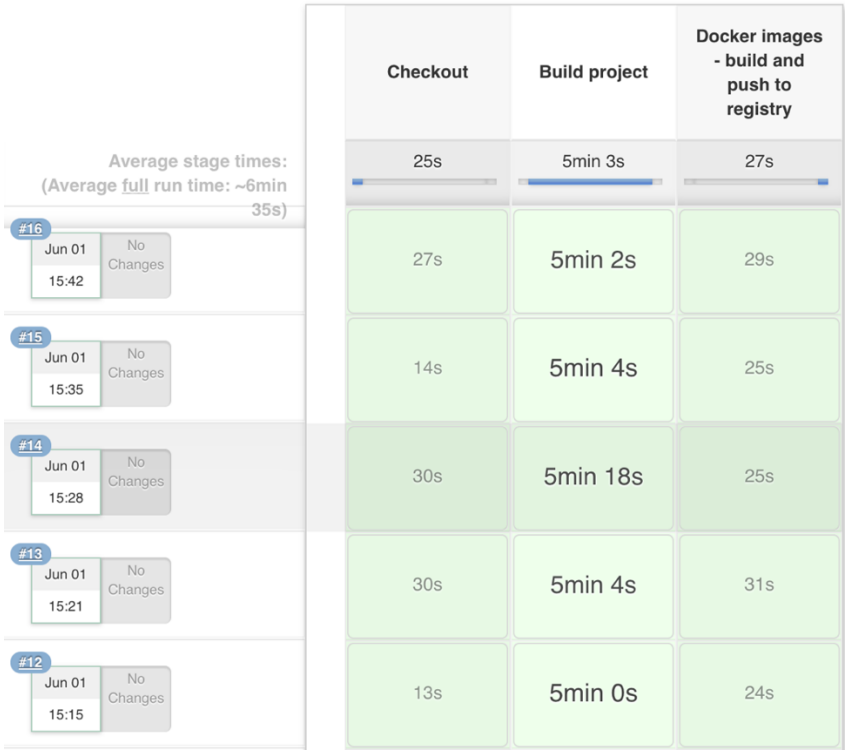

Figure 32 - Data Collected After Micro Frontends

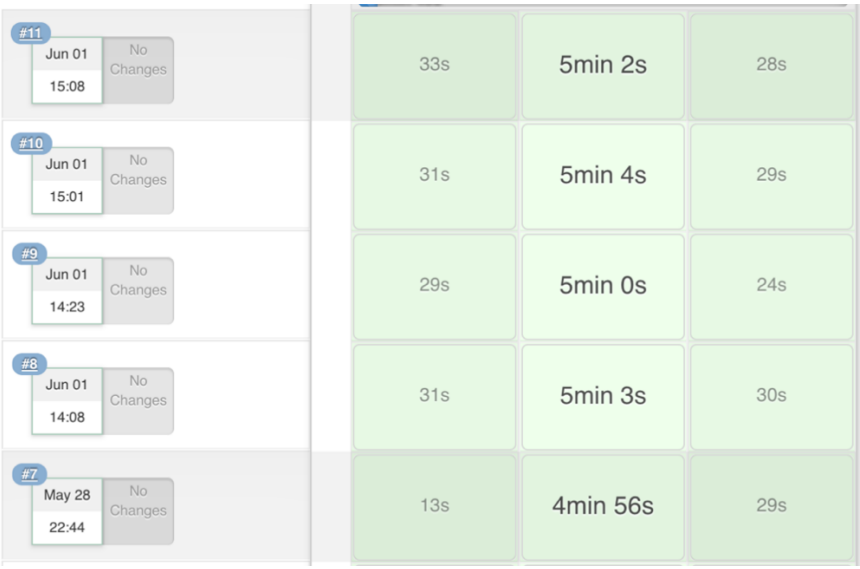

Figure 33 - Other Data Collected After Micro Frontends

Looking at the results obtained it is possible to see that the difference is a little bigger than 3 seconds. This difference is relatively small but shows a good indication that by following this approach it is possible to catch better time results.

5.2.2 Allow the use of several programming languages/several versions of the same programming language

The second objective was "Allow the use of several programming languages/several versions of the same programming language". In this case, this objective was associated with a single metric that defines, that at the end of this thesis those micro frontends developed should have different languages/versions.

Until the end of this thesis, at least 2 languages/versions must work together in the same application

As also said in chapter Solution, the final solution integrates a web component build in Angular 8 and a web component build in React 16 in an application that sits on implementation of Angular 7.

With such an analysis it is possible to conclude that this objective has been accomplished.

6 Conclusion

With this case study it was intended to make a change to a monolithic frontend of an i2S application, introducing the concept of micro frontends with a single API Gateway in it allowing to observe gains in the application.

Also, with this case study, it was possible to study ways of introducing this concept in frontends applications and understand the implications of each one. Additionally, to prepare for the implementation process, it was studied the JavaScript library React, that at that moment was unfamiliar to the author, unlike the Angular framework that was well known.

After this study and having the implementation approach defined, the implementation process was initiated. At that point the design of the application was detailed as well as the use cases, using UML annotation.

Having the design completed, the coding of the implementation started, where ways of separating the application were tested and by the end two of them were discarded.

The analysis of the separation process allowed to arrive at an approach considered the most appropriate, proven by the application of ATAM.

6.1 Critical Analysis

Based on the work that was made for this thesis, it is possible to conclude that the objectives were completed, like it is demonstrated in Table 9.

Table 9 - Objectives Accomplishment Overview

Objective	Accomplishment status	Comment
Propose a solution to replace a Frontend Monolith	Partial Completed	It is not possible to perform a complete micro frontend approach because of the team size
Allow the use of several programming languages/several versions of the same programming language	Completed	Used Angular and React

The introduction of the micro frontends concept allowed to observe some gains in the application. Although the complete integration of this concept does not make sense at the moment, it was still possible to prepare and document the necessary steps for this introduction using an isolated test case.

It is now also possible to know the necessary implications for the creation of micro frontends through the Web Components approach, both in Angular and React.

However, it should be noted that these two implementations have a completely different level of maturity.

While the Angular implementation is better structured and the code is optimized, the React implementation is less so. This is because Angular was already known by the author of the thesis and on the contrary React was learned in the course of it. That is, regarding the implementation in React, there may be flaws in the structuring of the code.

To fully perform a critical analysis of the work made, the following two tables, Table 10 and Table 11, present respectively an elucidation of the benefits and downsides, detailed in chapter Micro Frontend, of the implemented solution with comments related to each one of them.

Table 10 - Benefits Conclusions

Benefit	Comment
Incremental	Allows incremental upgrades, without any failure of the other pieces of the system
Customer Focus	Allows the allocation of a dedicated team with the only focus is the evaluation of the formulas
Source Code	Each micro frontend developed as only the dependencies that are required to fulfil their objectives in terms of functionality and user interface
Cross-functional teams	At the moment this was not introduced, but the team as all the tools necessary to do so
User interface	Allows the choice of the programming language that the team is more comfortable with
Fragments	The created micro frontend works without any dependencies from the other UI components and can be reused in any part of the code

Transitions between pages	Allows the render of the formula evaluation interface without any need of loading the entire page
Deployment	The micro frontends created have its isolated build pipeline

Table 11 - Downsides Conclusions

Downside	Comment
Payload size [3]	Initially the payload size was 2,83GB. With the introduction of the micro frontends, the payload increases to 3,61GB, because all the source code associated to them
Environment	Difficulty found along with the developing process, but for now, any solution was discovered
Managing development	The management of the development was easily done, considering the use case implemented but this problem will increase as more and more micro frontends are created
Consistency	Problem identified at an early stage but with fixes in the backend this no longer occur
Heterogeneity	Problem solved after the decision of only focusing on implementations based on Angular or React

6.2 Future Work

Although all the objectives of the master's work have been achieved, some improvements can be made and there is also room for some complementary work within the scope of the professional activity of the author of this document. In addition, the organization and its employees also have contributions to make in the coming months in order to achieve a satisfactory migration to a micro frontend approach.

Some aspects were irrelevant for the proof of concept implemented but they are necessary for the business context like making the implementation more dynamic. This dynamism is inherent to the possibility of the functionality described being used within

the context of the application, that is, requiring authentication, but also being used in an external context, not requiring authentication.

Once this is achieved, there will then be a general solution for any micro frontend created in the future. A way of providing the .css files in a location that they are accessible by authentication needs to be considered. By doing this it will be possible for the micro frontends created by the team or the micro frontends created by the clients to be used in them, thus maintaining the coherent interface. This should be done this way and not by making the same file available via HTTP, since following this second approach, there will not only be a need for the browser to make additional requests to obtain the data, but there will also be the introduction of another point of failure , if these requests fail for any reason.

Teams structures have to be think in order to facilitate the desired application architecture, a postulate that has been designate as the “inverse Conway maneuver” [45] and this aspect cannot be neglected in any company, and will be certainly considered in the future.

Finally, it is internally under discussion possible separations for the fragments to create those micro frontends.

6.3 Final Remarks

From a personal point of view of the author, this thesis helped him to:

- learn new technologies;
- improve the knowledge of the known technologies;
- systematize knowledge;
- learn new methods and processes related to document elaboration;
- learn different approaches to solve this thesis problem;
- document and analyse possible solutions.

7 References

[1] "5 technologies that made major impact on insurance industry," *HackerEarth Blog*, Mar. 29, 2018. https://www.hackerearth.com/blog/talent-assessment/5-technologies-insurance-industry/ (accessed Dec. 22, 2019).

[2] M. Hargrave, "Will Insurtech Disrupt the Insurance Industry?," *Investopedia*. https://www.investopedia.com/terms/i/insurtech.asp (accessed Feb. 08, 2020).

[3] M. Fowler, "Micro Frontends," *martinfowler.com*. https://martinfowler.com/articles/micro-frontends.html (accessed Dec. 28, 2019).

[4] Hashmap, "The What, Why, and How of a Microservices Architecture," *Medium*, Jun. 22, 2018. https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9 (accessed Jan. 25, 2020).

[5] "Micro Frontends - extending the microservice idea to frontend development," *Micro Frontends*. https://micro-frontends.org/ (accessed Jan. 25, 2020).

[6] G. T. Doran, "There's a S.M.A.R.T. way to write management's goals and objectives," *Manage. Rev.*, vol. 70, no. 11, 1981, [Online]. Available: https://community.mis.temple.edu/mis0855002fall2015/files/2015/10/S.M.A.R.T-Way-Management-Review.pdf.

[7] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, p. 131, Dec. 2008, doi: 10.1007/s10664-008-9102-8.

[8] "Active Reviews for Intermediate Design (ARID)." https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513472 (accessed Jan. 28, 2020).

[9] "Architecture Tradeoff Analysis Method Collection." https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513908 (accessed Jun. 24, 2020).

[10] W. Kenton, "How Cost-Benefit Analysis Process Is Performed," *Investopedia*. https://www.investopedia.com/terms/c/cost-benefitanalysis.asp (accessed Jan. 25, 2020).

[11] H. Harms, C. Rogowski, and L. Lo Iacono, "Guidelines for adopting frontend architectures and patterns in microservices-based systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, Paderborn, Germany, 2017, pp. 902–907, doi: 10.1145/3106237.3117775.

[12] "Microservices Pattern: API gateway pattern," *microservices.io*. http://microservices.io/patterns/apigateway.html (accessed Jun. 21, 2020).

[13] "Sam Newman - Backends For Frontends." https://samnewman.io/patterns/architectural/bff/ (accessed Jun. 20, 2020).

[14] "1 What Are Micro Frontends? · Micro Frontends in Action MEAP V03." https://livebook.manning.com/book/micro-frontends-in-action/chapter-1/v-1/ (accessed Jan. 25, 2020).

[15] "ThoughtWorks: A Global Software Consultancy." https://www.thoughtworks.com/ (accessed Jun. 21, 2020).

[16] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015.

[17] M. GEERS, "Micro Frontends - The Nitty Gritty Details or Frontend, Backend, Happyend." https://noti.st/naltatis/zQb2m5 (accessed Jan. 25, 2020).

[18] ENTANDO_TEAM, “entando.com - When To Use Micro Frontends.” https://www.entando.com/page/en/when_to_use_micro_frontends?contentId=BLG7254 (accessed Jan. 25, 2020).
[19] K. R. Müller, “Prototyping Micro Frontends,” *Medium*, Nov. 01, 2019. https://itnext.io/prototyping-micro-frontends-d03397c5f770 (accessed Jan. 25, 2020).
[20] Ö. Zafer, “Understanding Micro Frontends.” https://hackernoon.com/understanding-micro-frontends-b1c11585a297 (accessed Jan. 25, 2020).
[21] R. Gaur, “Breaking down the last Monolith - Micro Frontends,” *The DEV Community*. https://dev.to/aregee/breaking-down-the-last-monolith-micro-frontends-hd4 (accessed Jan. 26, 2020).
[22] K. Opperman, “Micro frontends — migrating legacy systems,” *Medium*, Dec. 19, 2017. https://medium.com/@kyle_77228/micro-frontends-migrating-legacy-systems-8a6a9d10d4d4 (accessed Jan. 26, 2020).
[23] C. Yang, C. Liu, and Z. Su, “Research and Application of Micro Frontends,” *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 490, p. 062082, Apr. 2019, doi: 10.1088/1757-899X/490/6/062082.
[24] KBall, “Microfrontends: the good, the bad, and the ugly,” *ZenDev, LLC*, Jun. 17, 2019. https://zendev.com/2019/06/17/microfrontends-good-bad-ugly.html (accessed Jan. 25, 2020).
[25] D. D. Toom, “My experience using micro frontends,” *Medium*, May 03, 2019. https://medium.com/passionate-people/my-experience-using-micro-frontends-e99a1ad6ed32 (accessed Dec. 23, 2019).
[26] OWASP, “Server-Side Includes (SSI) Injection | OWASP.” https://owasp.org/www-community/attacks/Server-Side_Includes_(SSI)_Injection (accessed Jan. 25, 2020).
[27] Estech Systems, “ESI Communication Server.” https://www.esi-estech.com/feature-commserver.html (accessed Jan. 25, 2020).
[28] Podium, “Podium · Easy server side composition of microfrontends.” https://podium-lib.io// (accessed Jan. 25, 2020).
[29] “Apache httpd Tutorial: Introduction to Server Side Includes - Apache HTTP Server Version 2.4.” https://httpd.apache.org/docs/current/howto/ssi.html (accessed Feb. 13, 2020).
[30] “Edge Side Includes | Customer Support | Akamai.” https://www.akamai.com/us/en/support/esi.jsp (accessed Feb. 13, 2020).
[31] “Introduction - webcomponents.org.” https://www.webcomponents.org/introduction (accessed Feb. 13, 2020).
[32] “What is AJAX? Front-End Developer, AJAX Programming,” *Hiring Headquarters*, May 05, 2015. https://www.upwork.com/hiring/development/how-ajax-works/ (accessed Feb. 13, 2020).
[33] “HTML | Iframes,” *GeeksforGeeks*, Nov. 27, 2017. https://www.geeksforgeeks.org/html-iframes/ (accessed Feb. 16, 2020).
[34] “Micro-frontend Architecture in Action with six ways. - DEV Community .” https://dev.to/phodal/micro-frontend-architecture-in-action-4n60 (accessed Dec. 22, 2019).
[35] “nginx.” https://nginx.org/en/ (accessed Feb. 20, 2020).

[36] “single-spa.” https://single-spa.github.io/single-spa.js.org/ (accessed Feb. 20, 2020).
[37] “3 Reasons You Should Avoid iFrames For Your Business Website,” *Tweak Your Biz*, Oct. 12, 2013. https://tweakyourbiz.com/technology/3-reasons-avoid-iframes-business-website (accessed Feb. 22, 2020).
[38] “Party | Insurance Glossary Definition | IRMI.com.” https://www.irmi.com/term/insurance-definitions/party (accessed Jun. 09, 2020).
[39] “Insured Party | legal definition of Insured Party by Law Insider.” https://www.lawinsider.com/dictionary/insured-party (accessed Jun. 09, 2020).
[40] “Sass vs. SCSS: which syntax is better?” http://thesassway.com/editorial/sass-vs-scss-which-syntax-is-better (accessed Feb. 09, 2020).
[41] “Direflow,” *Direflow*. https://direflow.io/ (accessed May 20, 2020).
[42] A. Remdt, “Handling data with Web Components,” *Medium*, Jun. 13, 2019. https://itnext.io/handling-data-with-web-components-9e7e4a452e6e (accessed Sep. 11, 2020).
[43] “API Gateway vs Backend For Frontend.” https://www.devdelly.com/api-gateway-vs-bff/ (accessed Sep. 10, 2020).
[44] A. Zalewski, *MODELLING AND EVALUATION OF SOFTWARE ARCHITECTURES*. 2013.
[45] M. Skelton and M. Pais, *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution, 2019.
[46] “Value Creation - strategy, organization, definition, school, company, business, competitiveness.” https://www.referenceforbusiness.com/management/Tr-Z/Value-Creation.html (accessed Feb. 22, 2020).
[47] Alexandra Twin, “Value Proposition: Why Consumers Should Buy a Product or Use a Service,” *Investopedia*. https://www.investopedia.com/terms/v/valueproposition.asp (accessed Feb. 22, 2020).
[48] “TAM SAM SOM - what it means and why it matters,” *The Business Plan Shop*, 15:05:00.0. https://www.thebusinessplanshop.com/blog/en/entry/tam_sam_som (accessed Feb. 22, 2020).
[49] A. Angular, “Angular.” https://angular.io/ (accessed Jan. 29, 2020).
[50] “What and Why React.js.” https://www.c-sharpcorner.com/article/what-and-why-reactjs/ (accessed May 19, 2020).
[51] “React – A JavaScript library for building user interfaces.” https://reactjs.org/ (accessed May 19, 2020).
[52] S. SMASHING BOXES, “Choosing a Front End Framework: Angular vs. Ember vs. React | Blog,” *Smashing Boxes*. https://smashingboxes.com/blog/choosing-a-front-end-framework-angular-ember-react/ (accessed Jan. 29, 2020).
[53] Microsoft, “Introduction · TypeScript.” http://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html (accessed Jan. 29, 2020).
[54] “An Introduction to CSS Pre-Processors: SASS, LESS and Stylus,” *HTML Mag*. http://htmlmag.com/article/an-introduction-to-css-preprocessors-sass-less-stylus (accessed Feb. 09, 2020).

[55] “Sass: Syntactically Awesome Style Sheets.” https://sass-lang.com/ (accessed Feb. 09, 2020).
[56] “Getting started | Less.js.” http://lesscss.org/ (accessed Feb. 09, 2020).
[57] “Expressive, dynamic, robust CSS — expressive, robust, feature-rich CSS preprocessor.” http://stylus-lang.com/ (accessed Feb. 09, 2020).
[58] R. L. Nord, M. R. Barbacci, P. Clements, R. Kazman, and M. Klein, “Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM):,” Defense Technical Information Center, Fort Belvoir, VA, Dec. 2003. doi: 10.21236/ADA421615.
[59] R. Schillinger, *Semantic Service Oriented Architectures in Research and Practice*. BoD – Books on Demand, 2011.

Annex A

Value Analysis

In this chapter, it is presented a value analysis of this project, where it is presented concepts like, value creation, value proposition and business model canvas that is related to the business area where this project fits.

Value Creation

Value creation is a set of actions that increase the worth of goods, services or even a business [46].

Value for the Customer

This thesis is contextualized in a problem solution for the insurance area. That solution aims to modernize the technological insurance system by recognizing user priorities like:

- Fast implementation: providing a 4-week time to market instead of the current nine-month;
- Reduced cost of compliance;
- Independence from current legacy systems;
- Easy to use systems;
- Configurable products.

Perceived Value

In a marketing perspective, the perceived value is the clients evaluation of the advantages of a product or service and its ability to accomplish their needs and expectations.

In nowadays, insurance companies spent almost three years with more than $500K cost to fully accomplish the desirable insurance product and its subscription lifecycle. So, the perceived value aims to reduce that costs by 50%, reaching approximately $250k.

Benefits

The significant benefit is that, by following this approach, the project team will be able to integrate components developed by other i2S teams allowing knowledge share across different teams. This characteristic not only brings value for i2S but also for its customers.

That kind of value is hugely crucial for customers because when using several i2S software, they will feel a sense of harmony in what they see, considering that some parts of the user interface are shared.

Sacrifices

One of the sacrifices of following a micro frontends approach is that teams lose some speed in their development process. This appends because despite all the benefits that this approach brings it also comes with a more complicated process of developing and integration. Of course, this only appends because the team that works on this project is smaller (three frontends and three backends), meaning that it is not possible the existence of several teams.

Value Proposition

A value proposition is a way of saying to customers, why they should do business in a company, instead of going to the competitors [47].

This thesis sits on a project that aims to cover the entire lifecycle of an insurance product definition, including the commercialization and its retirement, allowing a replacement of the insurance companies core system. This kind of process is extremely complex, so not every competitor can do the same.

Market

To do a market analysis, it is mandatory a calculation of some variables that help companies understand their market [48]. Those variables are TAM, SAM and SOM and in this case, will be populated with the number of insurance companies around the world.

- Total Available Market (TAM) is the total market demand for a product/service;
- Serviceable Available Market (SAM) is a portion of the TAM that is in companies geographical reach;

- Serviceable Obtainable Market (SOM) is a segment of the SAM that will be captured.

In this thesis context, the TAM is 3000. SAM is 1900, and SOM is 40. Figure 34 shows a visual representation of that data.

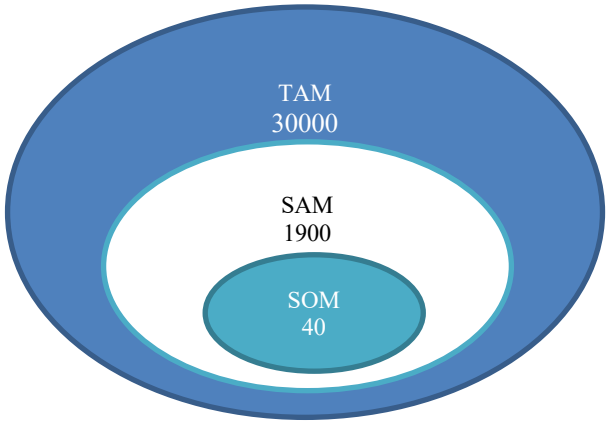

Figure 34 - TAM, SAM & SOM

Business Model Canvas

Business Model Canvas

Key Partners

- AWS;
- Microsoft;
- NOS Sistemas;

Key Activities

- Make proof-of-concepts and demos;
- Participate in events, fairs and conferences;
- Engage with integrators
- Be recognized as innovator;
- Promote content management;
- Implementation automation, continuous delivery/continuous integration

Key Resources

- Human resources (Software Developers);
- Customer network;
- Investment;

Value Propositions

- Cover the entire lifecycle of an insurance product definition, including the commercialization and its retirement
- Fast implementation of insurance products
- Decrease the cost of regulation compliance;
- Increase the number of addressable customer segments;

This kind of process is extremely complex, so not every competitor can do the same.

Customer Relationships

- Ease to use for first time users;
- Integration with customer system
- Chat bot assistance
- Helpline assistance
- Customer support cost increases as product spans over time zones

Channels

- Team of Business developers;
- Established Integrator Companies;
- Business Events
- Business Fairs
- Specialized conferences
- Specialized media

Customer Segments

- Tier 1 to 5 Insurance companies
- Reinsurers
- Brokers

Cost Structure

- Human resources: development, support and marketing and sales team
- Cloud (AWS, Google, Azure)
- Equipment: Workstations, office, support facilities
- Software Licenses

Revenue Streams

- Usage-based fee (pay-per-use)
- Modular approach (pay for component)
- Value-based pricing framed within usage capacity

Figure 35 - Business Model Canvas

Software competition

- Accenture Life Insurance & Annuity Platform;
- FLEXTERA Product Factory;
- Fineos Policy;

- TIA Solutions;
- eBao Tech Product Factory;
- Product Accelerator;
- Mind Tree Insurance;
- Pega Insurance Product Configurator.

Competitors

- Accenture;
- Diasoft;
- Fineos;
- TIA;
- eBaoTech;
- CSC;
- MindTree;
- Pega.

Competitive advantages and disadvantages

Advantages:

- Launching products into the market in a faster way;
- Compliance by design;
- Integration with clients software.

Disadvantages:

- Another solution to the customers;
    - A huge need to prove the value of the solution compared with the other solutions.

Annex B

Tools and Technologies

This chapter describes frameworks and tools usually used for frontend development.

Angular

Angular 2+ is a framework that allows the development of web and mobile applications, and it is written in Typescript [49].

This framework is based on components, which are characterized by a combination of an HTML template and a typescript class.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
export class AppComponent { name = 'Angular'; }
```

Figure 36 - Component Example
Source: [13]

In Figure 36 it is demonstrated a basic example of the combination mentioned before, which is associated with a typescript class to an HTML file to generate the result demonstrated in Figure 37.

Figure 37 - Result Obtained
Source: [13]

Each component starts with "@Component", which describes how HTML, CSS and the class are related in order to obtain the desired result.

The "template" property defines, in this specific example, a message embedded in a header. This message starts with "Hello" and ends with "{{name}}". While the application is running, this framework replaces "{{name}}" with the value of the variable "name" defined in the class through an interpolation that allows relating expressions present in the HTML template with properties defined in the component.

Each of the Angular applications has files with a specific function that evolves as the application grows. Files outside the "/ src" directory have the function of building the application. On the contrary, those in this directory are the ones that belong to the application and that serve to highlight all the components defined within it. Three essential files will be outside this directory:

- app.component.ts: main component, where all other components will be listed;
- app.module.ts: specifies the application of how it will be built;
- main.ts: allows the application to be compiled, so that it can be executed in any browser.

Through the command line of Angular, called "Angular-CLI", a set of commands is available that allows the launch of an Angular application, where the files referred to before are automatically created. Here, too, it is possible to generate components, policies, services. It is through this that when generating components/directives, they are added directly to the file "app.module.ts". Thus, the user is not concerned with the integration of the created components, but with their correct functioning.

Currently, "Node.js" and "npm" are essential tools for developing applications in Angular and other platforms. Where the first provides the client with several development tools and the second has several JavaScript libraries extremely useful for the development of this type of application.

Moving to the architecture of this platform, it is possible to identify eight main blocks:

- Components;
- Templates;
- Metadata;

- Data binding;
- Directives;
- Services;
- Dependency injection.

In the Figure 38 a diagram is presented that represents the relationships between the blocks.

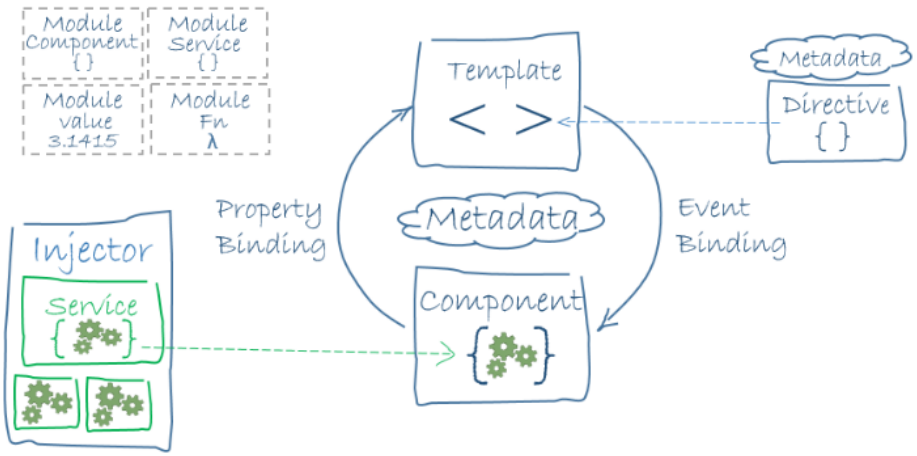

Figure 38 - Main Blocks of an Angular Application
Source: [13]

Modules

All Angular applications have at least one module class, which by convention is called "AppModule". It is here that all the declarations of the components, directives and pipes are specified. Here references are made to services and components in order to be available to the entire application. Finally, this is also where the main component is defined. Figure 39 shows an example of what was mentioned previously.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Figure 39 - Angular Module

Source: [13]

Components

As previously mentioned, a component controls a portion of the screen called a "view". Here, the components logical behaviour is defined so that a correct interaction with the "view" occurs.

Templates

A template is a way for HTML to communicate with Angular, describing how the component should behave. It is also possible to use another component inside another one using HTML tags that reference that same component.

Data Binding

It allows coordination between parts of the template with parts of the component. This coordination can occur in both directions using the "ngModel" directive.

Services

Angular services are typically a class with a well-defined purpose, which will be used by the components where they were defined.

Metadata

Metadata is the way to tell Angular how to process the class. This is possible through "@Component".

- selector: informs Angular that where the selector is found, it must load the template of the component where it was defined;

- templateUrl: allows the developer to address the template without having to build it in the class itself;
- providers: define the services that the component needs.

Dependency Injection

Way to provide a new instance of a class with the dependencies it requires. In most cases, these dependencies are related to services.

React

React is another open-source JavaScript library used to build interfaces for single-page applications and like Angular, it helps create reusable UI components [50].

Declarative

Allowing the design of simple and declarative views for each state, that render the right components when data change, the developed code become more predictable and easier to debug [51].

Component-Based

Construct encapsulated components that maintain their state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, it is easier to pass valuable data through the application [51].

Simple Component

React components implement a render() method that takes input data and returns what needs to be displayed [51]. This behaviour is showed in Figure 40.

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  document.getElementById('hello-example')
);
```

Figure 40 - React Example of Render Method
Source: [15]

Stateful Component

A React component is also able to access input data and also can maintain the component internal state data allowing the rendered mark-up to be updated by re-invoking the render() method [51].

External Plugins

React also allows the use of other libraries and frameworks. In Figure 41 is presented as an example of the use of and an external Markdown libraries, to convert the <textarea> value in real-time [51].

```
class MarkdownEditor extends React.Component {
  constructor(props) {
    super(props);
    this.md = new Remarkable();
    this.handleChange = this.handleChange.bind(this);
    this.state = { value: 'Hello, **world**!' };
  }

  handleChange(e) {
    this.setState({ value: e.target.value });
  }

  getRawMarkup() {
    return { __html: this.md.render(this.state.value) };
  }
```

Figure 41 - External Library
Source: [15]

Single-Way data flow

In React, components renderer receives immutable values as properties in HTML tags. The component itself is not able to modify any of these properties but can pass a call back function to do such modifications, like it is showed in Figure 42 [50].

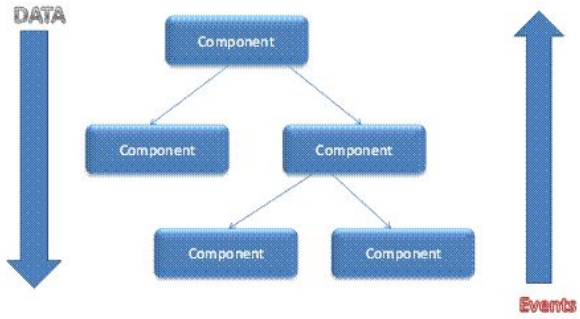

Figure 42 - Process of "properties flow down; actions flow up"
Source: [15]

Virtual Document Object Model

An in-memory data structure cache that computes the changes that occur in HTML and updates the browser. This characteristic allows a particular feature that enables the programmer to code as if the entire page is rendered on each change whereas React only renders components that truly change [50].

A comparing study of two frameworks was made [52]. The results of this study are shown in Table 12 and Table 13.

Table 12 - Pros/Cons Angular

Pros	Cons
Higher performance compared to the previous version	Little documentation, as it is a new platform
Rendering on the server-side	Hard to find resources
Native graphical interface	Complexity in the directives
Most popular platform	Many interactive elements can be slow
Easy to test	Complexity in integration
Platform with all the necessary tools for building applications	Difficult to debug
Supports new languages	

It allows the user not to have to focus on component integration, but on creating a proper application	
Code reuse	

Table 13 - Pros/Cons React

Pros	Cons
Performance	Architecture different from that to which most users are used to using it
Rendering on the server-side	Very sophisticated presentation layer
Native graphical interface	
Simple	
Accessible libraries	
Supports new languages	
Faster updates	
Code reuse	

After a careful analysis of these platforms, it was possible to reach several conclusions:

React

- Lighter in terms of occupied memory;
- Appropriate choice if it is necessary to modernize the base code.

Angular

- The most popular;
- Choice of choice by large companies.

Typescript

TypeScript is an open-source programming language, characterized by being a superset of JavaScript, compiles for pure JavaScript and is the language recommended for those

who will use the Angular platform. It provides tools auto-completion and navigation as referred to in the tables on the next page [53].

The fact of having reliable tools of IntelliSense and code reconstruction has a significant impact on the code writing process, characteristics that make TypeScript the choice for the development of presentation applications. One of the advantages of using it is the fact that it has a "cleaner" code than JavaScript, thus allowing a better understanding of what that code does. When compiling TypeScript, JavaScript code is automatically generated, even in the possibility of compilation errors, allowing it to be interpreted by most current browsers.

CSS

It is a language that controls the presentation of HTML pages.

In bigger projects, maintaining the primitive CSS it is harder, so to improve the implementation process, developers build a set of pre-processors that helped to achieve the reusable writing, maintainable, extensible codes in CSS and decrease the amount of code in a project. Those pre-processors extend CSS with variables, operators, interpolations, functions, mixins [54].

SASS [55], LESS [56] and Stylus [57] are the most common pre-processors.

HTML

It is an annotation language that allows the creation of Web pages through symbols and codes that are later interpreted by the browser, defining the structure and formatting of a page.

Annex C

Methods for Designing High-level Software Architecture

As mentioned in sub-chapter "Approach and development process", there are three methods for designing high-level software architecture that could be useful for this thesis, ARID, ATAM and CBAM. So, in this sub-chapter, those three will be detailed.

ATAM

Architecture tradeoff analysis method, also known as ATAM [58], is probably the most effective method to perform an architecture evaluation and in its base consists in four phases: Presentation of the architecture, Investigation and analysis, Testing and finally Reporting [59].

Presentation

This phase starts with an introduction of the business side, follow up with a presentation of the architecture that is the target of the analysis.

Investigation and Analysis

In this phase, the chief software architect explains the general approach objectively without any evaluation hints and includes a high-level architecture view of the styles and patterns used.

Essentially, what this phase output is a construction of a "Utility tree" which compress a representation of the critical qualities attributes in the evaluation project and specific scenarios that should be prioritized before the analysis.

Also, in this phase, must occur a matched comparison of the architectural approach and the "Utility tree" so that the suitability of the strategy is following the elicited requirements.

Testing

This phase requires a big audience to perform a crosscheck of the scenarios elaborated int the "Utility tree", which consequently will lead to some modifications.

The number of possible scenarios can be extremally huge, but three kinds of them are the most important.

1. Use cases scenarios oriented to the functionalities of the system;
2. Growth scenarios that represent future projections of the software under evaluation;
3. Exploratory scenarios that represent unrealistic wort case of the growth scenarios.

Reporting

In this last phase, must occur a presentation of the evaluation results obtained.

ARID

The Active Reviews for Intermediate Designs, also known as ARID, follows a different approach of the previously mentioned method [59].

In this case, it is not necessary the existence of a complete architecture, but instead, be used in partial designs of the architecture [8].

Mainly, this method is divided into two phases.

Rehearsal

This is the first phase of this approach, and it can only be carried out by the chief architect and the review leader. In this phase, it is demonstrated a presentation of the part of the architecture that was previously prepared and rehearsed by one of the intervening actors mentioned before.

After this, both of the participants prepared sample scenarios that will serve as a base for the following phase and in this process are already made the organizational preparations also for the next phase.

Review

This phase, that is the main phase of this method starts with a presentation of the ARID method, followed by the presentation developed in the previous phase.

Since the reviewers are all developers, is asked to them to implement the most crucial scenario of the architecture in terms of pseudo-code.

During this step, the intervening actors do not give any support to the developers until they realize that they are stuck or that they enter in a wrong path. This last step is repeated several times to finally gather the results.

CBAM

The Cost Benefit Analysis Method is concerned with the economic implications of the architecture types, which contrast with the methods previously-mentioned that focus on technical criteria evaluation [59].

This method requires preparative steps given by an iteration of ATAM, to collects different alternatives of the architecture design and calculates costs and benefits of each one. Those obtained values are related by a measure that is called desirability and afterwards, they are ranked according to that measure [58].