Politecnico di Torino

Politecnico di Torino
Faculty of engineering

Master's degree thesis in Computer Engineering

Exploring Software Architectural Transitions: From Monolithic Applications to Microfrontends enhanced by Webpack library and Cypress Testing

Advior:
Prof. Fulvio Corno

Candidate:
Maria Akl

Graduation session of July
Academic year 2023/2024

Abstract

This thesis delves into the software architecture's dynamic evolution, with a particular focus on the paradigm shift from monolithic applications to the world of microservices and, subsequently, microfrontends. It analyses into the root causes driving the migration away from monolithic architectures, particularly examining the challenges and rewards associated with this transition. This analysis sheds light on the transformative power of microservices in providing agility, scalability, and resilience that are crucial qualities for building successful software in today's dynamic world.

Building upon this foundation, the thesis invesitgates the evolution journey from microservices to microfrontends. It explores the strategic advantages of decoupling user interfaces from backend services. This architectural shift empowers independent development, deployment, and scalability that are critical factors in optimizing the modern front-end experience. Notably, the thesis examines the instrumental role played by the Webpack library in facilitating the implementation of microfrontends. Its capabilities in module bundling, code splitting, and dynamic loading are explored in detail, providing valuable insights into this modern technology.

Furthermore, the thesis recognizes the importance of testing in the context of microfrontends, discussing the unique challenges and considerations involved in ensuring the quality and reliability of these front-end components. The Cypress testing framework is presented as a powerful tool to confront these challenges, and its functionalities are explored to shed light on effective testing strategies.

To solidify these theoretical concepts, the thesis culminates in a comprehensive analysis of a real-world case study. This practical application offers invaluable insights and practical guidance for software architects, developers, and organizations grappling with the complexities of software architecture evolution. By fostering a deeper understanding of the principles and techniques underpinning the transition from monolithic applications to microservices and microfrontends, stakeholders gain the power to leverage these modern architectural paradigms.This helps them to build robust, resilient, and maintainable software systems capable of thriving in the ever-growing demands of the digital age.

Contents

List of Figures

List of Tables

1 Introduction: Charting the Evolving Landscape of Software Architecture

Software architecture, the guiding framework that defines a system's structure and behavior, is not a rigid blueprint. Instead, it's a living entity constantly adapting to the ever-changing world of technology and business needs. This ongoing process, known as software architecture evolution, is crucial for the development of software systems in the modern world.

This evolution is driven by various factors, including technological advancements, changing business requirements, and lessons learned from past experiences. Understanding the concept of software architecture evolution is crucial for software engineers and architects to adapt to the ever-changing landscape of technology and deliver robust, scalable, and maintainable systems.

The story of software architecture evolution begins in the early days of computing with monolithic architectures. We could think of these as monolithic cathedrals ,that are impressive structures where all functionalities reside under one roof. While this approach offered simplicity in development and deployment, its inflexibility became a major bottleneck as applications grew in complexity.

As computing technology marched forward and applications grew in size, the limitations of monoliths architectures became undeniable.This paved the way for alternative paradigms like Service-Oriented Architecture (SOA). SOA offered a modular approach, where functionalities were broken down into loosely coupled services that could communicate and collaborate, it an be compared to building a city block by block, allowing for more independent development and deployment of individual services. However, SOA also had its drawbacks, particularly in terms of complexity in managing service interactions and communication protocols.

From this emerged the idea of microservices architecture, the current core of the software development world. Microservices take the modularity of SOA a step further, breaking down applications into even smaller, independent services with well-defined APIs. We can Imagine it as a collection of tiny, specialized shops instead of a single department store each shop focuses on its own product, collaborating loosely with others to provide a complete smooth customer experience.

The growing popularity of microservices paved the way for microfrontends. Microservices, by decomposing backend functionalities, introduced the idea of independent development and deployment for different parts of an application. Microfrontends took this concept a step further, applying it to the user interface (UI) itself. While microservices focus on backend logic, microfrontends focus on the frontend presentation layer. Imagine a complex web application with microfrontends, the UI can be broken down into smaller, independent, and technology-independent components. These components, can be developed, deployed, and updated independently, mirroring the benefits of microservices for the backend. This approach promotes faster UI development cycles, improves maintainability, and the ability to leverage the best frontend technologies for each specific component.

On another hand there are some technological advancements, such as the proliferation of cloud computing, containerization, and DevOps practices, that have enabled new architectural patterns and paradigms to emerge. For example, the advent of virtualization and container orchestration platforms like Docker and Kubernetes has facilitated the adoption of microservices architecture by providing scalable infrastructure for deploying and managing distributed systems.

Furthermore, changing business requirements and market dynamics necessitate adaptations in software architecture to remain competitive.Organizations must be agile and responsive to customer needs, which often requires architectural changes to support rapid development, deployment, and iteration of software systems.

Overall it is safe to say that software architecture evolution is not a one-time event but rather a continuous process of improvement and refinement. It involves iteratively assessing the current state of the architecture, identifying areas for enhancement or optimization, and implementing changes incrementally. This iterative approach to architecture evolution aligns with agile software development principles, where feedback loops and continuous integration and deployment practices enable teams to adapt quickly to changing requirements and market conditions. By understanding the drivers of evolution and embracing a the idea of continuous improvement, software engineers and architects can design architectures that are resilient, scalable, and future-proof, enabling organizations to thrive in today's rapidly changing technological landscape.

2 From Monolithic to Microservices: A Paradigm Shift

2.1 Monolithic applications

Monolithic applications represent a traditional approach to software architecture where all components of an application are tightly integrated into a single, cohesive unit . This type of architecture provides a developpment enviroment where all functionalities of an application (user interface,business logic, data access) reside within a single, unified codebase. We can think of it as a single large server that houses everything the application needs to run. This centralized codebase simplifies the initial development process, as developers have a clear view of the entire application's structure, and it is usually a good choice at the start of the project [41, 15]. With a single database, transactions are usually easy to handle, as most database systems provide ACID (Atomicity, Consistency, Isolation, Durability) transactions.

Thanks to the characteristics mentioned previously, the process of deploying the application is narrrowed down to a single simple action. Additionally, monolithic architectures often leverage shared memory and resources, facilitating seamless communication between components and efficient resource utilization.

But as applications grow in size, so does its complexity. Therefore, a typical way to handle complexity in monolithic architecture is to split the application into different layers. This layered approach, known amongst developers, is widely used in networking and operating systems [30]. The Model-View-Controller (MVC) architecture, a fundamental software design principle, organizes an application into three interconnected components: Model (data management), View (user interface), and Controller (business logic). This structure ensures a clear separation of concerns, enhancing modularity and maintainability. As shown in Figure 1, the application is organized into distinct layers, each with a specific responsibility:

- **Presentation Layer**: Handles the user interface and user experience aspects.
- **Business Logic Layer**: Contains the core functionality and business rules of the application.
- **Data Access Layer**: Manages interactions with one database that handles all the data that is related to this application.

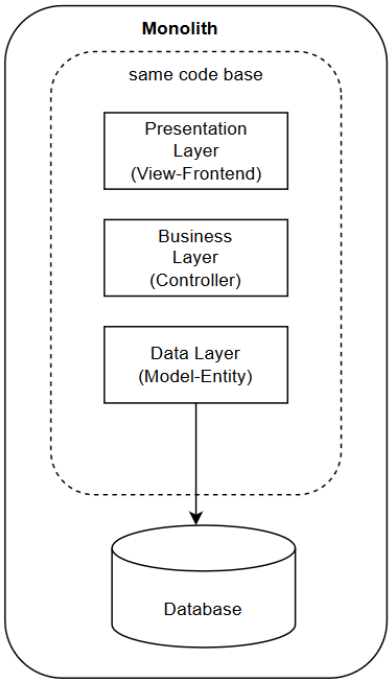

Figure 1: Architecture of a layered monolithic application

This approach makes development, deployment, and scaling easy when the size of the application is moderately small. However, as the size of the application and organization grows, so do the different layers of the application. Unless a lot of attention is paid to the architecture and quality of the codebase, it is very likely that the quality of the different layers deteriorates.

The deterioration happens because of business requirements that force developers to make solutions which are not optimal. These sub-optimal solutions should be refactored but the time for refactoring can be hard to get, which leads to short-term solutions becoming long-term solutions. As the size of the codebase grows and the quality of the codebase deteriorates, it becomes harder to add newfeatures and modify old features because the developer has to find the correct place to apply these changes, thus resulting in slower development cycles.

On the other hand, the deployment always contains every part of the application. So when only one component is changed, the whole application has to be re-deployed. This makes continuous deployment hard. Clear modular boundaries inside a monolithic codebase can be hard to achieve and maintain during the development. Programming languages gives some tools for developers to ensure the modularity and loose-coupling in the monolithic codebase, but breaking these boundaries is easy because developers can change the visibility with ease and thus break the modularity.

Problems are also encountered in testing, where large layered monolithic application can run into problems with the CI pipeline. The build times of CI pipelines can become longer as there is a lot of code to compile and thousands of automated tests to be run.

These issues can lead to a situation where the organizations are too scared to deploy their applications continuously. Even though there are big challenges with using CD with monolithic codebase, there are examples of companies making it possible such as Etsy [22].

As monolithic applications matured and their complexity grew, limitations inherent to the architecture began to surface. Scaling monolithic applications became cumbersome, requiring the entire codebase to be replicated even if only specific features needed increased capacity. Tight coupling between components made modifications and updates difficult, potentially introducing bugs or regressions. As highlighted in table 1, the introduction of layers within a monolithic application has led to some improvements, yet it still retains certain limitations. In response to these limitations, the concept of microservices emerged. This architectural style decomposes applications into smaller, independent services, each responsible for a specific business function.

To sum it all up, we can say that monolithic applications represent a traditional approach to software architecture characterized by tightly integrated components deployed as a single unit. While monolithic architectures offer simplicity and ease of deployment, they also come with challenges related to scalability, flexibility, and agility. Understanding the characteristics and trade-offs of monolithic architectures is essential for organizations evaluating their architectural choices and seeking to build scalable, resilient, and maintainable software systems.

Feature	**Monolithic Application**	**Monolith Layered Application**
Codebase	Single, unified code	Single, unified code
Structure	Unstructured	Layered (presentation, business logic, data access)
Advantages		
Development	Simple development, deployment	Improved maintainability compared to unstructured monolith
Disadvantages		
Scalability	Scalability issues	Can still have scalability issues
Tight Coupling	Tight coupling between components	Tight coupling to an extent
Maintenance	Maintenance challenges	Improved maintainability compared to unstructured monolith but still complicated with larger applications
Use Cases		
	Simple applications, rapid prototyping	When some level of organization is needed in a monolith

Table 1: Brief overview of Monolithic and Monolith Layered Architectures

2.2 Foundations and Features of Microservices Applications

Microservices architecture is a software development approach,which has gained popularity in the last few years , that structures an application as a collection of loosely coupled, independently deployable services. Each service is focused on a specific business capability and can be developed, deployed, and scaled independently, allowing for greater flexibility and agility in the development process. A rule of thumb about the size of a microservice could be that it can be rewritten in two weeks.These services communicate with each other through well-defined interfaces, typically using lightweight protocols such as HTTP or messaging queues (provided by each service) [9, 28]. Let us make a small example of an order application that follows the microservices architecture: one microservice could handle the creation of orders and another microservice could then handle the creation of an invoice that relates to the order. Figure 2, illustrates this example where the frontend of the application calls those two services: these two services have their separate codebase sand if there is a need to communicate between them, the communication is done through the APIs [22].

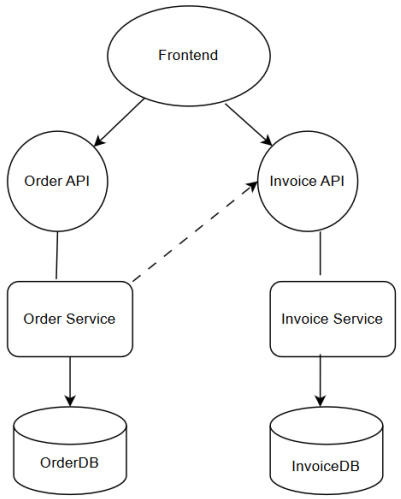

Figure 2: Example of a small microservice application

Here are some of the fundamental features that are central to microservices architectures[40]:

1. **Service Decoupling**
   One of the fundamental principles of microservices architecture is service decoupling. Services are designed to be independent of each other, with minimal dependencies between services. It means that services should not know anything about the internals of other services.This allows development teams to work on individual services without being constrained by the implementation details of other services. Service decoupling promotes modularity, flexibility, and autonomy, enabling organizations to evolve and scale their applications more effectively.

2. **Single Responsibility**
   Microservices should always comply with the Single Responsibility Principle (SRP). SRP means "gathering the things that change for the same reason and separating those things that change for different reasons". Each microservice is responsible for a specific business capability or functionality. This principle of single responsibility ensures that services are focused and cohesive, with clear boundaries and well-defined interfaces. By encapsulating specific business logic within each service, organizations can achieve better separation of concerns and maintainability, making it easier to understand, test, and evolve individual services independently.

3. **Independent Deployment**
   Microservices are designed to be independently deployable, allowing organizations to release updates and new features more frequently and efficiently. Since each service is deployed separately, development teams can iterate on features and fixes without impacting the entire application.

Independent deployment also enables organizations to adopt continuous delivery practices, where changes are automatically deployed to production as soon as they are ready, reducing time-to-market and improving agility.

4. **Polyglot Architecture**
   Microservices architectures embrace the concept of polyglot programming, allowing organizations to use a diverse set of programming languages, frameworks, and technologies for different services.Teams developing microservices can make independent choices from other teams depending on their business and technical challenges. Some limitations should be in place,in order to limit the number of languages in the application.But this flexibility enables teams to choose the best tools and technologies for each service based on its specific requirements and constraints. Polyglot architecture promotes innovation and productivity, as teams can leverage the strengths of different technologies to solve complex problems more effectively.

5. **Scalability and Elasticity**
   Microservices architectures are inherently scalable and elastic, allowing organizations to scale individual components based on demand. Since services are deployed independently, organizations can allocate resources more efficiently and handle varying workload conditions effectively. For example,one service might require horizontal scaling while another one requires vertical scaling. Horizontal scaling means adding more instances that serve the microservice, while vertical means adding more capacity to the instance: hence it is possible to scale every service separately depending on their need. This enables better resource utilization and performance compared to monolithic architectures, where scaling often involves replicating and deploying multiple instances of the entire codebase.

6. **Resilience and Fault Isolation**
   Microservices architectures facilitate fault isolation, allowing failures in one service to be contained and managed without affecting the entire application. This enhances overall system resilience and reliability, as organizations can recover from failures more quickly and minimize downtime. By isolating failures to individual services, organizations can implement targeted mitigation strategies and maintain high availability even in the face of failures.

7. **Continuous Integration and Deployment**
   Microservices architectures promote continuous integration and deployment practices, where changes are automatically tested, built, and deployed to production environments as soon as they are ready. This enables organizations to release updates and new features more frequently and reliably, reducing the risk of introducing bugs or regressions. Continuous integration and deployment foster a culture of rapid experimentation and innovation, enabling organizations to iterate and evolve their applications more effectively.

Because of everything that microservices has to offer, companies such as Amazon, LinkedIn and Netflix have made the transformation. Their positive experiences and long-term usage with this architecture style have caught the interest of many other companies and developers interested in new architecture styles These companies are very open about their development processes and, for example, Netflix has open-sourced a lot of their internal tools .

2.3 Motivations Behind Migrating from Monolithic Applications to Microservice

The transition from monolithic applications to microservices is driven by a variety of significant motivations. These motivations encompass a range of factors that address the limitations of monolithic architectures and leverage the advantages offered by microservices [25, 22]. Key among these motivations are:

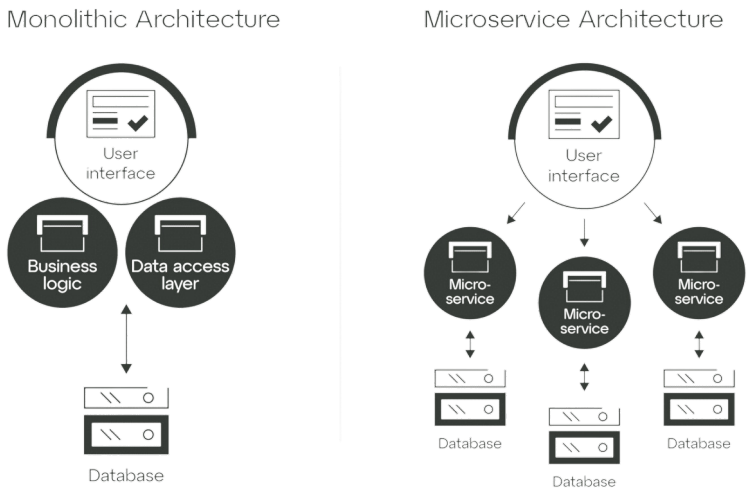

Figure 3: Monolithic vs Microcervices architecture

1. **Scalability**
   Monolithic applications often face challenges in scaling due to their tightly coupled nature. Scaling a monolithic application typically involves replicating and deploying multiple instances of the entire codebase, regardless of which specific components experience increased demand. This can lead to inefficient resource utilization and higher infrastructure costs. In contrast, microservices architectures enable organizations to scale individual components independently based on demand. By breaking down applications into smaller, independently deployable services, organizations can allocate resources more efficiently and handle varying workload conditions effectively.

2. **Agility and Flexibility**
   Monolithic architectures can hinder agility and innovation, as changes to the application require coordination across multiple teams and can be slow and cumbersome. Microservices architectures, on the other hand, promote agility and flexibility in software development. By breaking down applications into smaller, independently deployable services, development teams can iterate more rapidly and release updates without impacting the entire application. This enables organizations to respond quickly to changing market demands, experiment with new features, and innovate more effectively.

3. **Resilience and Fault Isolation**
   Monolithic applications are vulnerable to failures that can impact the entire application. A failure in one component of a monolithic application can cascade and affect other components, leading to system-wide outages. Microservices architectures facilitate fault isolation, allowing failures in one service to be contained and managed without affecting the entire application. This enhances overall system resilience and reliability, as organizations can recover from failures more quickly and minimize downtime.

4. **Technology Diversity**
   Monolithic applications are often built using a single technology stack, which can limit the flexibility and innovation potential of the application. Microservices architectures enable organizations to adopt a diverse set of technologies and programming languages for different services. This flexibility allows teams to choose the best tools and technologies for each service, optimizing performance and productivity. Additionally, microservices architectures facilitate polyglot persistence, allowing organizations to use different databases and data stores based on the specific requirements of each service.

5. **Continuous Delivery and Deployment**
   Monolithic architectures can complicate the continuous integration and deployment processes due to their intertwined components, making it difficult to implement frequent updates without extensive testing. Microservices, by contrast, support continuous delivery and deployment

practices more effectively. Each service can be developed, tested, and deployed independently, which accelerates the release cycle and reduces the risk of introducing bugs into the system. This approach enables faster time-to-market for new features and improvements.

6. **Organizational Alignment**
   Monolithic applications can create bottlenecks within organizations due to their centralized and interconnected structure. This often necessitates larger, cross-functional teams, which can lead to coordination challenges and slower decision-making processes. Microservices architectures allow for smaller, autonomous teams that can manage individual services independently. This alignment of development teams with specific business capabilities enhances collaboration, speeds up development processes, and improves overall productivity.

2.4 Migration Strategies from Monolithic to Microservices Applications

Migrating from monolithic to microservices applications is a complex and multifaceted process that requires careful planning, execution, and coordination. Several common migration strategies have emerged to help organizations navigate this transition effectively [26]. We will mention some of the migration strategies and techniques:

2.4.1 Domain-Driven Design (DDD)

Domain-Driven Design (DDD) [32] is a software development methodology that prioritizes the accurate modeling of business domains and the establishment of distinct boundaries between them. When transitioning from monolithic applications to microservices, DDD plays a crucial role in identifying and defining the bounded contexts of these applications. These bounded contexts can then be developed as independent microservices. This alignment ensures that each microservice is focused, cohesive, and has clear responsibilities with well-defined interfaces.

Moreover, employing DDD facilitates better communication among development teams and business stakeholders by using a common language tailored to the specific domain. This common understanding helps in reducing misinterpretations and increases the efficiency of the development process.

2.4.2 Strangler Fig Pattern

The Strangler Fig Pattern, a concept introduced by Martin Fowler, offers a strategic approach to migrating monolithic applications towards a microservices architecture. Unlike a "big bang" rewrite, which can be risky and disruptive, the Strangler Fig Pattern promotes a gradual and controlled transition. It involves progressively replacing components of the monolithic application with newly developed microservices over time [7]. This incremental approach allows organizations to:

- **Minimize Risk:** By introducing new features and functionalities as microservices, the existing functionality within the monolith remains operational. This minimizes the risk of introducing bugs or regressions that could impact the entire application during the migration process.
- **Iterative Refinement:** As new microservices are developed and take over specific responsibilities, developers can refactor and decompose the remaining parts of the monolith in a controlled manner. This enables continuous improvement and ensures a smooth transition to the new architecture.
- **Phased Rollout:** The gradual replacement allows for phased rollouts, where specific functionalities can be migrated to microservices and tested independently. This phased approach minimizes disruption to users and allows for course correction if necessary.
- **Maintain Business Continuity:** Existing business functionality continues to operate throughout the migration process. This minimizes downtime and ensures a seamless transition for end-users.

As shown in the figure 4, the "strangler fig" metaphor describes the process. Just as a strangler fig vine gradually envelops a host tree, microservices progressively take over responsibilities from the monolith, eventually leading to its complete replacement.

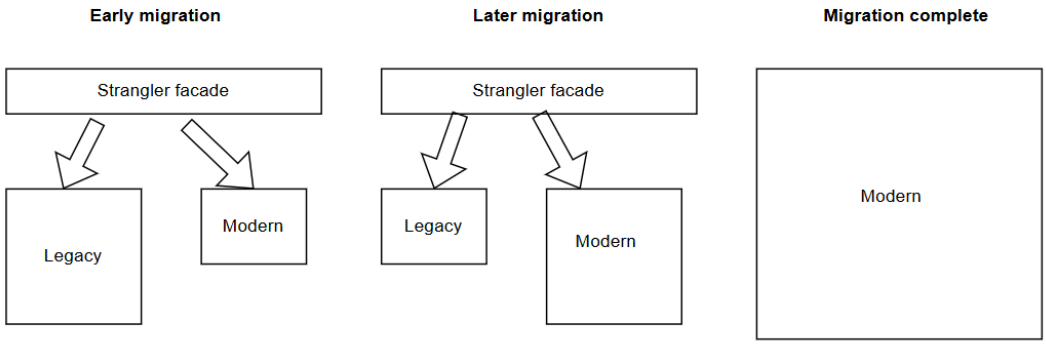

Figure 4: Strangler fig pattern

2.4.3 API Gateway

An API Gateway is a centralized component that acts as a single entry point for client applications to access microservices. In the context of migrating from monolithic to microservices applications, an API Gateway can serve as a bridge between the monolithic application and microservices architecture. By exposing APIs that encapsulate functionality from the monolithic application and routing requests to the appropriate microservices, organizations can gradually transition functionality from the monolithic application to microservices without disrupting existing client applications.

2.4.4 Extract-Transform-Load (ETL) Process

The Extract-Transform-Load (ETL) approach provides a structured method for migrating functionalities from monolithic applications to microservices architectures in a controlled and incremental manner. This strategy minimizes risk and disruption by focusing on specific components. During the ETL process, developers first identify and extract cohesive sets of functionalities that represent well-defined business capabilities. These functionalities are then refactored and transformed into independent microservices, potentially adapting code and logic to ensure proper communication within the microservices architecture. Finally, the newly developed microservices are integrated and "loaded" into the existing system, establishing communication channels and ensuring smooth data flow

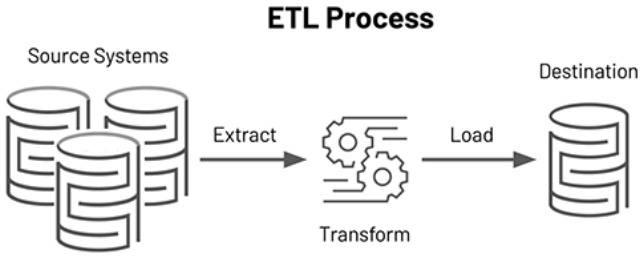

Figure 5: ETL process

2.4.5 Parallel Development

Parallel Development involves building new features and functionalities as microservices alongside the existing monolithic application. This approach allows organizations to iteratively migrate functionality from the monolithic application to microservices while continuing to deliver value to customers. By enabling parallel development of monolithic and microservices-based functionality, organizations can minimize risk and accelerate the migration process.

Domain	Patterns or approaches	Migration report	Article
Energy services	strangler pattern and DDD	increased performance and reliability	[29]
Cargo delivery	DDD	with growing granularity coupling and cohesion decreases, DDD can help finding ideal granularity	[52]
Derivatives Management System	gradual migration, possibly strangler pattern	reduced time to market	[16]
Automotive Problem Management System	gradual migration, possibly strangler pattern	difficulties due to complicated tight coupling	[16]
Automotive Configuration Management System	strangler pattern	significantly improved time to market	[16]
Retail Online Shop 2	greenfield pattern and DDD	culture of the development changed to more open and unconstrained	[16]

Table 2: Overview of concrete migration cases, migration report contains brief information, either a migration result or note about the process

The table 2, showcases several application migrations, detailing the employed patterns and approaches, along with feedback regarding the migration outcomes [45]. Across these examples, it's evident that the majority of migrations yielded positive results, benefiting both the business operations and performance levels.

By selecting the appropriate migration strategy based on their specific requirements and constraints, organizations can minimize risk, accelerate the migration process, and realize the benefits of microservices architectures.

2.5 Challenges faced during the migration process

Transitioning from a monolithic architecture to a microservices architecture introduces a multitude of challenges that organizations must adeptly manage to ensure a smooth and successful transformation. This migration process is intricate and multifaceted, involving not only significant technical reconfigurations but also substantial changes to organizational workflows, development methodologies, and operational strategies [37]. Successfully navigating these challenges, that are briefely discussed below, is crucial for leveraging the full benefits of microservices, such as enhanced scalability, flexibility, and resilience."

1. **Complexity and Interdependence**
   A significant challenge in migrating to microservices is managing the inherent complexity and interdependence of monolithic applications. Monolithic systems often have tightly coupled components, making it difficult to identify and extract cohesive functionality sets to implement as microservices. Untangling dependencies and establishing clear boundaries between microservices necessitates meticulous analysis and refactoring, which can be both time-consuming and error-prone.

2. **Data Management and Consistency**
   Microservices architectures pose challenges related to data management and consistency. Unlike monolithic applications where data is typically stored in a single database, microservices architectures involve each service having its own database or data store. This can lead to data duplication, inconsistency, and synchronization issues. Organizations must adopt robust data

management strategies, such as event sourcing or distributed transactions, to maintain data integrity and consistency across microservices.

3. **Service Communication and Orchestration**
   Effective communication and orchestration between services are crucial in microservices architectures. As the number of services increases, managing inter-service communication becomes more challenging. Organizations need to implement reliable communication patterns and protocols, such as RESTful APIs or message queues, to facilitate efficient and dependable service interactions. Additionally, designing and implementing mechanisms for handling retries, timeouts, and circuit breaking is essential to ensure system reliability and performance.

4. **Operational Complexity**
   Compared to monolithic applications, microservices architectures introduce additional operational complexity. Managing numerous services requires robust monitoring, logging, and orchestration tools to ensure system reliability and performance. Organizations must invest in infrastructure and tooling to effectively support the operational demands of microservices. Furthermore, independently deploying and scaling services necessitates automation and DevOps practices to streamline deployment processes and minimize downtime.

5. **Organizational Change and Culture**
   Migrating to a microservices architecture often entails significant organizational change and cultural transformation. Development teams must embrace new working methodologies, such as cross-functional teams, agile practices, and DevOps principles, to efficiently manage and operate microservices-based systems. Additionally, fostering a culture of collaboration, experimentation, and continuous improvement is essential for fully adopting the principles and practices of microservices architectures.

3 Emergence of Microfrontends

3.1 Different approaches of building graphical user interfaces

Currently, there are three primary approaches to building GUIs in web applications: Server-Rendered Pages, Single Page Applications (SPAs), and Single Page Applications with Chunk Splitting:

- **Server-Rendered Pages** utilize template engines, where the markup is typically embedded within controllers on the back end [50]. This approach generally results in a mostly static GUI, although dynamic elements can be injected and loaded as separate static JavaScript files. When a user requests a page, the back end processes this request, generates the necessary HTML, and delivers the fully rendered page along with the required resources to the client. One significant aspect of this method is that it does not require front-end development into a distinct team. Consequently, this often results in a less collaborative development process, as the same team handles both the back-end and front-end aspects. This unified approach can help development in some cases, but it may also limit the potential for parallel workflows. The setup for generating these server-rendered pages is typically straightforward, making it easy to implement initially. However, if developers choose to extensively customize the template engine, the setup can become more complex and require additional effort. Moreover, maintaining these pages can present significant challenges. Templates may be distributed across various parts of the system, making it difficult to locate the correct template when issues arise or updates are needed. This distributed nature of templates can complicate the maintenance process, as developers must navigate through different parts of the code to identify and correct the appropriate template, potentially leading to increased development time.

- **Single Page Applications (SPAs)** [33] are built entirely with JavaScript and are delivered to the user as a single, bundled .js file. This approach provides a highly dynamic and interactive user experience, eliminating the need for full page reloads as users navigate through the application. SPAs are relatively easy to set up, largely due to the availability of boilerplate templates and frameworks that simplify the initial development process, such as : React, Angular, Vue.js and

many more. However, while SPAs offer significant advantages in terms of user experience and initial setup simplicity, they also present certain challenges. One major challenge is the difficulty in distributing development work across different teams. Since the entire application is contained within a single JavaScript bundle, it can be hard to seperate clear boundaries for different teams to work on independently. This can result in collaborative development efforts and make it harder to scale the development process. Moreover, maintaining large SPAs can become increasingly complex over time. As the application grows and more features are added, the single JavaScript bundle can become quite large, leading to potential performance issues and longer load times. Additionally, managing a large codebase within a single file can be cumbersome, making it harder to debug, test, and update the application. Developers need to employ strategies such as code splitting, lazy loading, and modular design to mitigate these issues, but these solutions can add additional layers of complexity to the development and maintenance process.

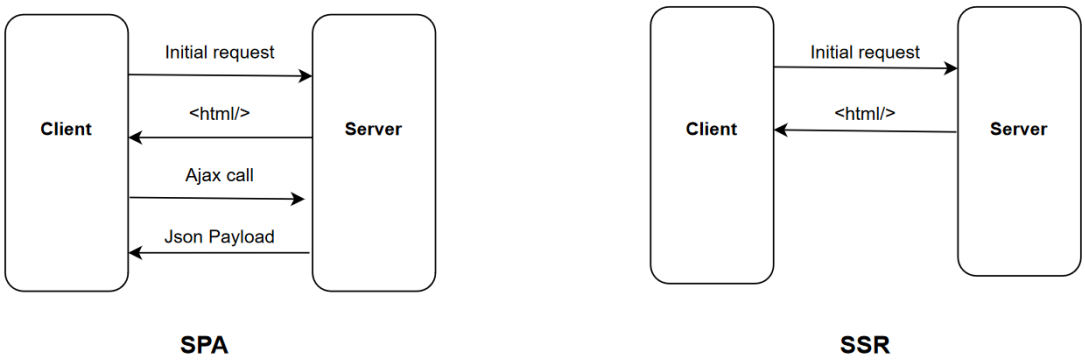

Figure 6: Difference between single page and server side applications

- **Single Page Applications with Chunk Splitting** operate similarly to standard SPAs, with the difference being that the build output consists of multiple files that are loaded on demand. These files contain subsets of the application's functionality, which helps to speed up startup time and improve the overall user experience. By breaking down the application into smaller chunks, it reduces the initial load time, making the application more responsive. This approach also optimizes resource usage by only loading necessary code, which can lead to better performance and reduced bandwidth consumption. Additionally, chunk splitting can facilitate easier debugging and maintenance by isolating specific parts of the codebase.

Finally we arrive to the primary focus of this thesis that is on **Micro-Frontends Architecture**, an advanced approach built on top of Single Page Applications (SPAs) [27]. This method seeks to address the limitations inherent in traditional SPAs by enabling different teams to work on distinct, independently deployable parts of the application. This modularity facilitates better scalability and maintainability, as each team can develop, test, and deploy their components in isolation.

The following table summarizes the four methods of developing GUIs along with their key characteristics, providing a comprehensive overview [31]:

Domain	Server-Rendered Pages	Single Page Application	SPA with Chunk Splitting	Micro-frontends
Static or dynamic GUI	Typically static with dynamic parts	Typically dynamic	Typically dynamic	Typically dynamic
Framework restrictions	Without framework	Single framework	Single framework	Any number of frameworks
Delivery to the client	Loads one page at a time	Loads fully, in the beginning	Loads partially, on request	Loads partially, on request
Simultaneous work	Possible	Usually, impossible	Usually, impossible	Possible
Setup complexity	Simple	Simple	Normal	Complex
Maintenance complexity	Normal	Normal/Complex	Normal/Complex	Normal/Complex

Table 3: Comparison of Approaches to Building Front-end

3.2 Microfrontend core

Micro frontends, represent a paradigm shift in the architecture and development of web applications, addressing the challenges in modern software engineering [57, 10, 20, 19]. At its essence, the concept of micro frontends, as shown in figure 7, adresses the fragmentation of monolithic frontend applications into smaller, self-contained, and independently deployable units, each encapsulating specific functionalities or features. This modularization fosters improved organization and maintainability, empowering development teams to concentrate on distinct business domains or individual user interface components. The idea of modularity is throughout the idea of micro frontends, stressing the significance of decomposing large and monolithic frontend codebases into granular components, whether they be widgets, modules, or entire pages. Embracing a modular approach grants organizations greater flexibility in managing and evolving their frontend architectures, facilitating swifter iterations, simpler testing procedures, and enhanced collaboration between frontend and backend teams. Furthermore, micro frontends streamline the development and deployment of frontend components by allowing teams to work autonomously and release updates without being constrained by dependencies on other parts of the application. This decoupled development model fosters agility and innovation, empowering teams to experiment, iterate, and respond swiftly to evolving requirements or market trends. Additionally, the adoption of micro frontends encourages the adoption of best practices in software engineering, such as continuous integration and continuous deployment (CI/CD), enabling seamless integration and delivery of updates across different parts of the application.

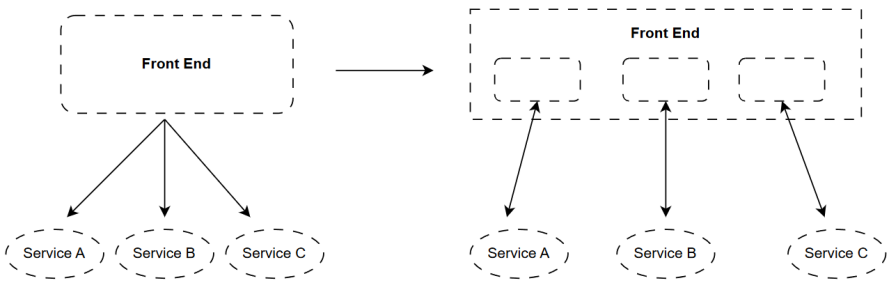

Figure 7: Microservices architecture with monolithic frontend conversion to micro-frontends

3.3 Relationship to microservices architectures

The relationship between micro frontends and microservices architectures is symbiotic, as both concepts share similar principles and complement each other in building scalable, modular, and maintainable software systems [31, 17]. Microservices architecture focuses on decomposing backend systems into smaller, independently deployable services, each responsible for a specific business capability. Similarly, micro frontends extend this decomposition to the frontend layer, breaking down monolithic frontend applications into smaller, self-contained units, each owning a distinct user interface component or feature. This parallel decomposition enables organizations to align frontend and backend development efforts, fostering better separation of concerns and enabling teams to work autonomously on specific business domains. Furthermore, micro frontends and microservices architectures both promote modularity, independence, and flexibility, allowing organizations to scale development efforts more effectively and respond rapidly to changing requirements or market demands. By adopting a polyglot approach to both frontend and backend development, organizations can leverage a diverse set of technologies, frameworks, and programming languages, optimizing each component for specific requirements and constraints. The integration of micro frontends and microservices architectures requires careful orchestration and communication between frontend and backend teams, ensuring that frontend components align with corresponding backend services and adhere to common standards and protocols. Additionally, both architectures emphasize the importance of composition and integration, enabling organizations to combine and orchestrate frontend and backend components seamlessly, providing users with a unified and consistent experience across different devices and platforms. In summary, micro frontends and microservices architectures are complementary approaches to building scalable, modular, and maintainable software systems, with each concept reinforcing and enhancing the principles and practices of the other. Together, they empower organizations to deliver high-quality user experiences efficiently and innovate with agility and confidence in the rapidly evolving landscape of modern software development.

3.4 Challenges of using microfrontends

Navigating into the world of micro frontends brings a set of unique challenges in modern web development. While these architectural patterns promise many benefits, they also introduce complexities that require thoughtful considerations. From ensuring smooth integration between various frontend components to managing dependencies and versioning, micro frontends present a series of challenges. We will therefore explore the challenges of adopting this architecture [2]:

1. **Increased Complexity:** Micro frontends add complexity due to the management of numerous independently deployable components. This complexity arises from coordinating updates, ensuring compatibility, and maintaining a consistent user experience. For example, imagine a large e-commerce platform composed of multiple micro frontends handling product browsing, cart management, and checkout. Coordinating changes across these micro frontends while ensuring a seamless user experience requires careful planning and coordination.

2. **Cross-Cutting Concerns:** Coordinating cross-cutting concerns such as routing, authentication, and internationalization across micro frontends can be challenging. For instance, implementing a consistent authentication mechanism across multiple micro frontends requires shared authentication services or libraries. Ensuring that users remain authenticated as they navigate between different micro frontends enhances the overall user experience and security.

3. **Versioning and Dependency Management:** Managing versioning and dependencies between micro frontends is crucial to prevent compatibility issues and regressions. For example, if one micro frontend relies on a specific version of a shared library, updating that library may require changes in multiple micro frontends. Implementing versioning strategies, such as semantic versioning, and using dependency management tools like npm or yarn, helps streamline this process.

4. **Performance Overhead:** Micro frontends can introduce performance overhead due to increased network latency and resource consumption. Loading multiple frontend components dynamically can lead to longer initial page load times and increased memory usage. Employing techniques

like lazy loading, code splitting, and caching helps mitigate these performance issues and improve the overall user experience.

5. **Testing and Quality Assurance:** Testing and quality assurance become more challenging in micro frontend architectures. Each frontend component must undergo comprehensive testing, including unit testing, integration testing, and end-to-end testing. Tools like Jest, Cypress, and Selenium aid in automating tests across different micro frontends, ensuring functionality and compatibility

6. **Developer Experience:** Adopting micro frontends can impact developer experience as developers work with multiple codebases, technologies, and frameworks. Maintaining consistency in coding standards, tooling, and processes is essential to ensure productivity and collaboration. Providing documentation, training, and sharing best practices fosters a positive developer experience and promotes knowledge sharing.

7. **Deployment and Operations:** Deploying and managing micro frontends in production environments requires robust deployment pipelines, monitoring, logging, and orchestration tools. Implementing continuous integration and continuous deployment (CI/CD) pipelines automates the deployment process, while monitoring tools like Prometheus and Grafana provide insights into application performance and health

3.5 Advantages of adopting a microfrontend architecture

In this subsection, we'll briefly mention the key benefits of adopting a microfrontend architecture, highlighting its capacity to streamline development workflows, promote team collaboration, and drive innovation in the digital landscape while mentioning some brief examples in each case [2, 18]:

1. **Modularity:**

   - **Enhanced Organization:** Microfrontend architecture facilitates the breakdown of large frontend applications into smaller, self-contained units, improving organization and management. For instance, in an e-commerce platform, each microfrontend could handle a specific section such as product listings, cart management, or checkout.
   - **Improved Maintainability:** With modular and isolated frontend components, maintenance becomes easier. For example, updating the checkout process in an e-commerce application can be done independently without impacting other parts.
   - **Reusability:** Microfrontend architecture promotes the reuse of UI components across different sections of an application. For instance, a custom dropdown menu component developed for the product listing section can be reused in the checkout process.

2. **Independent Development and Deployment:**

   - **Autonomous Teams:** Microfrontend architecture empowers frontend teams to work independently on specific features or functionalities. For example, a team responsible for the product search functionality can develop and deploy updates without waiting for other teams.
   - **Faster Time-to-Market:** Decoupling frontend development and deployment accelerates the release of updates and new features. For example, an online news platform can quickly roll out a breaking news feature without affecting other parts of the website.

3. **Polyglot Frontend:**

   - **Technology Flexibility:** Microfrontend architecture allows organizations to use a variety of frontend technologies based on their needs. For example, a media streaming service may use React for its video player component and Vue.js for its chat feature.
   - **Language Agnostic:** Microfrontend architecture supports the use of different programming languages or frameworks within the same application. For instance, a travel booking platform may use Angular for its booking flow and Svelte for its interactive map feature.

4. **Scalability and Performance:**

    - **Granular Scaling:** Microfrontend architectures enable organizations to scale frontend components independently based on demand. For example, a social media platform can scale its messaging feature separately from its newsfeed component.
    - **Optimized Resource Utilization:** With microfrontend architecture, resources are utilized efficiently by loading only necessary components. For instance, a productivity application loads collaboration tools only when users access them, reducing initial load times.

5. **Faster Iterations and Releases:**

    - **Continuous Delivery:** Microfrontend architecture supports continuous delivery practices, allowing organizations to release updates frequently and reliably. For example, a project management tool can deploy bug fixes and feature enhancements seamlessly.
    - **Iterative Development:** Microfrontend architecture enables organizations to iterate on frontend components based on user feedback. For instance, an e-learning platform can continuously improve its course navigation based on user engagement metrics.

6. **Better User Experience:**

    - **Personalization:** Microfrontend architecture enables organizations to personalize user experiences based on preferences and behaviors. For example, a music streaming service can recommend personalized playlists based on listening history.
    - **Responsive Design:** With microfrontend architecture, organizations can create responsive interfaces that adapt to different devices. For instance, a weather application displays information differently on desktop and mobile devices for optimal viewing experience.

3.6 Different Approaches to Microfrontend Implementation

There is different approaches to implementing microfrontends, each offering unique advantages and considerations: we will give an overview of three distinct methodologies: from composition at build time to runtime composition and the backend for frontend approach, we navigate through the landscape of microfrontend implementation strategies, providing insights into their application and relevance in modern web development.

3.6.1 Composition at Build Time

Composition at build time is a significant aspect of microfrontend architecture, reshaping how applications are developed and organized [44]. With this method, each microfrontend is build as an independent module, focusing on specific functionalities to enhance user experience. Imagine a large e-commerce platform looking to upgrade its online presence. Different microfrontends, designed to manage distinct areas such as product displays, shopping cart handling, and checkout processes, form the foundation of this digital ecosystem.

Frameworks like React or Angular provide the ways for building these microfrontends. Each frontend team operates independently, creating their microfrontend with attention to detail. Product displays are carefully curated, shopping cart functionalities are optimized for smooth transactions, and checkout processes are streamlined to guide users seamlessly through the purchase journey in our example.

During the build phase, tools like Webpack or Module Federation Plugin plays an important role. They orchestrate the compilation, bundling, and optimization of code. Webpack combines them into a unified package as it is shown in figure 8. This package represents the collective effort of various frontend teams, harmonizing their contributions into a cohesive whole.

Once completed, this integrated package showcases the power of composition at build time. Deployed to the server, it is ready to engage users with its intuitive interfaces and reliable performance.

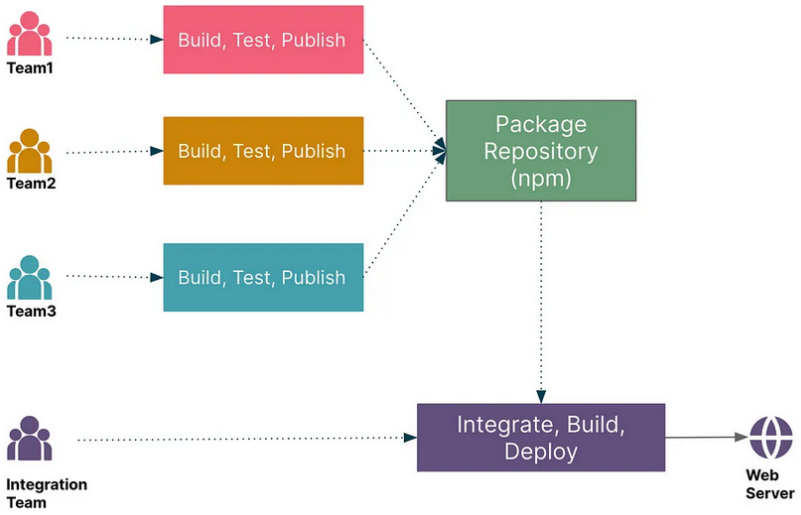

Figure 8: Composition at build time

3.6.2 Backend For Frontend (BFF):

Backend for Frontend (BFF) is a crucial aspect of microfrontend architecture, redefining how backend services are tailored to meet the diverse needs of frontend components. In this approach, each microfrontend has its own dedicated backend server, known as the Backend for Frontend, which provides the necessary data and services specific to that microfrontend. Picture a comprehensive content management system (CMS) where different microfrontends handle various aspects such as user management, content creation, and analytics.

Frameworks like Node.js or Spring Boot serve as the backbone for developing these Backend for Frontend services, empowering backend teams to focus on the unique requirements of each microfrontend. User management services are fine-tuned to handle authentication and authorization, content creation services are optimized for efficient storage and retrieval of data, and analytics services are tailored to process and analyze user interactions.

During development, each microfrontend is paired with its corresponding Backend for Frontend service, ensuring tight integration and seamless communication between frontend and backend components. This close coupling enables frontend teams to access the necessary data and services effortlessly, enhancing development efficiency and reducing dependencies on other backend systems. Upon deployment, each Backend for Frontend service operates independently, serving as a dedicated gateway for its associated microfrontend.

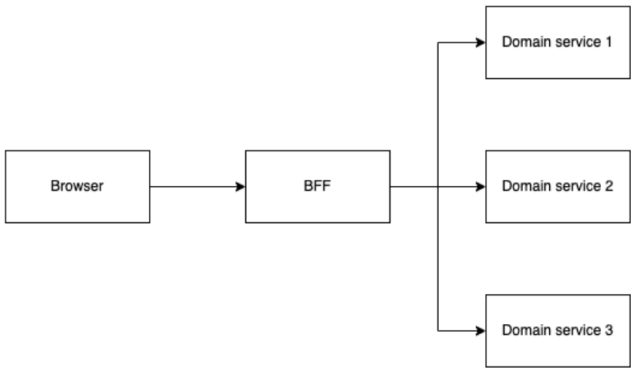

Figure 9: Backend for frontend

3.6.3 Composition at run Time

Composition at runtime is another vital aspect of microfrontend architecture, altering how applications are structured and delivered [5]. In this method, each microfrontend operates independently and is fetched and composed dynamically when the application is loaded in the browser. Imagine a sophisticated dashboard application with various widgets representing different microfrontends responsible for displaying diverse data such as charts, tables, and graphs. These microfrontends are deployed separately, and when a user accesses the dashboard, a client-side orchestrator fetches and combines them on-the-fly based on the user's preferences and permissions.

Frameworks like React or Vue.js provide the foundation for developing these microfrontends, allowing frontend teams to work autonomously on their components. Each team focuses on fine-tuning their microfrontend to ensure optimal performance and user experience. Charts are crafted to present data in a visually appealing manner, tables are optimized for easy navigation and data presentation, and graphs are designed to convey complex information with clarity.

During runtime, specialized JavaScript orchestrators handle the composition of microfrontends, fetching and integrating them seamlessly into the main application. This dynamic composition enables flexibility and agility, as microfrontends can be updated and deployed independently without disrupting the entire application. The result is a dynamic and responsive user interface that adapts to the user's needs and preferences in real-time.

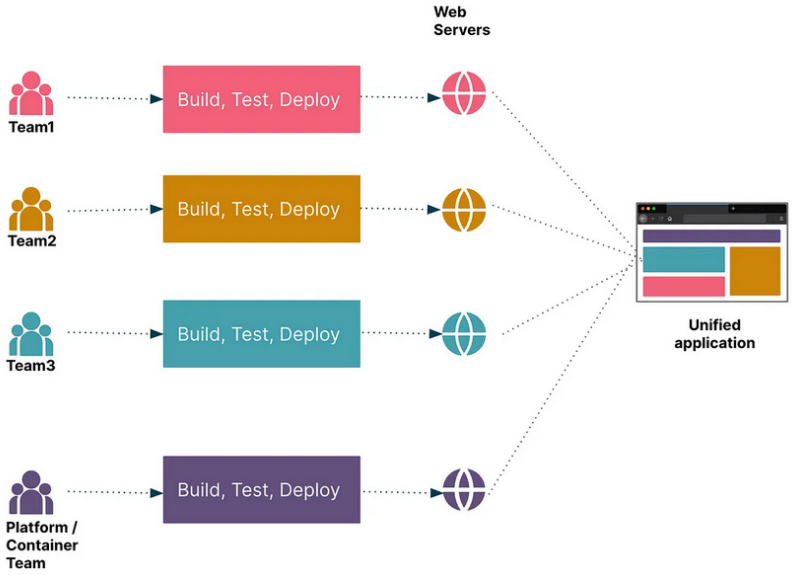

Figure 10: Composition at Run Time

After a brief overview of the three microfrontend methods, the following table 4, summarizes their key characteristics. In the next few sections, we will explore the first method, composition at build time, in greater detail, focusing on the capabilities of the Webpack library

Aspect	Composition at Build Time	Composition at Run Time	Backend for Frontend (BFF)
Description	Microfrontends are integrated into a single application during the build process.	Microfrontends are composed and integrated in the browser during runtime.	A custom backend is created for each frontend, serving specific data and business logic.
Performance	Generally faster since the composition is done ahead of time.	Might introduce latency as components are fetched and composed on the client side.	Performance depends on the efficiency of the backend and the number of network requests.
Flexibility	Less flexible; any change requires a new build and deployment.	Highly flexible; changes can be made independently and deployed immediately.	Moderately flexible; changes in the backend or frontend can be done independently.
Complexity	Lower complexity in deployment but higher in the build process.	Higher complexity in managing runtime dependencies and state.	Adds complexity by introducing an additional backend layer.
Isolation	Limited isolation; shared code and dependencies are bundled together.	Better isolation; each microfrontend can be loaded and executed independently.	High isolation; frontends and their corresponding backends are loosely coupled.
Scalability	Scaling might be challenging due to the tight coupling at build time.	Highly scalable as each microfrontend can be scaled independently.	Scalability depends on both the frontend and the backend scaling strategies.
Use Case	Suitable for applications with infrequent updates and where performance is critical.	Ideal for applications needing frequent updates and independent deployments.	Best for applications requiring specialized backend logic for different frontends.

Table 4: Overall summary of Microfrontend Approaches

4 The Webpack library

4.1 Introduction to the library

Webpack [6], introduced in 2012, has become an indispensable tool for managing the complexities of modern JavaScript applications. It tackles the challenge of organizing code into reusable modules, handling dependencies between them, and ultimately delivering optimized bundles for browsers. Originally created by Tobias Koppers, Webpack's evolution has been remarkable. It has grown from a basic bundler to a powerful platform supporting various build configurations, loaders for transforming different file types, and a vast plugin ecosystem to extend its functionality.

Webpack takes modules with dependencies and generates static assets representing those modules. It helps to manage and bundle the dependencies of an application, ensuring that the final bundle includes all the necessary code in the correct order. This process improves the performance and maintainability of web applications.

4.2 Primary Advantages of Using Webpack

Webpack has become an important part of modern JavaScript development, offering a powerful suite of features that streamline your workflow and enhance the performance and maintainability of your applications. Let's delve deeper into the key reasons why Webpack is an essential tool in the development toolbox:

- **Modularization Made Easy**

  Webpack promotes a modular approach to coding, where your application is broken down into smaller, self-contained modules. This improves code organization significantly. Imagine a large, monolithic codebase where it's difficult to understand, modify, and reuse specific functionalities. Webpack allows you to break it down into logical modules, each responsible for a specific task. This makes your codebase easier to navigate, maintain, and scale as your project grows.

- **Dependency Management Nirvana**

  Gone are the days of manually managing dependencies between modules. Webpack takes the burden off our shoulders by automatically resolving dependencies. It analyzes the code, identifies all the modules it relies on, and ensures they are included in the final bundle. This eliminates the risk of missing dependencies that could cause errors in the application. Webpack also intelligently handles version conflicts, ensuring compatibility between different libraries and frameworks.

- **Performance Optimization Champion**

  Webpack is a like champion for performance optimization. It offers features like minification, which reduces the size of the code by removing unnecessary characters and whitespace. This translates to faster loading times for the users. Additionally, Webpack utilizes code splitting, a technique that breaks the application into smaller chunks. The browser only loads the chunks that are currently needed by the user, further improving initial load times and overall performance. Webpack also employs tree-shaking, an advanced optimization technique that eliminates unused code from the final bundle, resulting in a leaner and more efficient application.

- **Asset Bundling for a Streamlined Workflow**

  Webpack doesn't just handle JavaScript – it can also bundle various assets like CSS, images, and fonts alongside the code. This simplifies the deployment process and reduces the number of HTTP requests your browser needs to make. Imagine having separate files for your JavaScript code, stylesheets, and images. Webpack can bundle them together, resulting in fewer requests and a smoother user experience.

- **Loader Power: Transforming File Types with Ease**

  Webpack's loader system empowers you to work with various file types seamlessly. Loaders are essentially plugins that can transform different file formats before they are included in the final bundle. For instance, you can use a loader to convert SCSS files into standard CSS or compile TypeScript code into JavaScript. This extends Webpack's capabilities and allows you to leverage modern features and frameworks in the projects without limitations.

- **Plugin Ecosystem: A World of Possibilities**

  Webpack boasts a vast and vibrant plugin ecosystem. These plugins act as extensions, further enhancing Webpack's functionality. we can find plugins for tasks like code analysis, image optimization, code linting, and more. This allows us to tailor Webpack to the specific project needs and streamline the development workflow.

- **Flexibility at our service**

  Webpack offers a high degree of customization through its configuration options. WE can fine-tune the bundling process to meet the project's specific requirements. Whether it's controlling how modules are bundled, setting up code splitting strategies, or configuring loaders and plugins, Webpack provides the flexibility to craft a build process that perfectly suits the type and necessities of the application.

Considering the complexity of our application and the need for features like code splitting and tree shaking, it's essential to choose a bundler that abides to these requirements. The following table compares Webpack and other bundlers like Browserify, and Rollup to help us make an informed decision, and we can therefore confirm that for larger and more complex applications Webpack is the better choice to go for:

Feature	Webpack	Browserify	Rollup
Focus	Module bundling, complex applications	Legacy code, simple bundling	Library bundling, small bundles
Strengths	Flexible, powerful, large community	Simple to use, good for ES5	Small bundles, designed for libraries
Weaknesses	Steeper learning curve, complex configuration	Limited functionality, not ideal for modern features	Less mature ecosystem, might not suit complex apps
Technology	Webpack Module Federation, SSG	Browserify transforms, require statements	Rollup plugins
Dependency Management	Automatic	Manual	Automatic
Code Splitting	Yes	Limited	Yes
Tree Shaking	Yes	No	Yes
Loader Support	Extensive	Limited	Limited
Plugin Ecosystem	Vast	Limited	Growing
Configuration	Complex (flexible)	Simple	Simpler than Webpack
Use Cases	Complex modern JavaScript applications	Legacy code, simple projects	Libraries, small utility applications

Table 5: Comparison of JavaScript Bundlers

4.3 Understanding the Mechanics of Webpack

4.3.1 Core Concepts and Webpack Configuration

Understanding how Webpack functions requires an understanding of several key concepts [53, 55, 51] that form the foundation of its operation. These core principles are essential to comprehend how Webpack manages and optimizes the various modules and assets within a project. While discussing these concepts, we will have a brief overview of the composition of the `webpack.config.js` file that serves as the central configuration hub for Webpack. It defines how Webpack should process the application's code and assets :

Entry Points: Entry points are the starting points for Webpack's bundling process. These are the JavaScript files that kick off its internal dependency graph. We can define multiple entry points, allowing to structure the application in a modular fashion. Each entry point will be bundled into a separate file. Typically, the entry point is the application's main JavaScript file. Webpack uses this as the starting point to determine which modules and libraries are needed to create the final bundle. Examples of entry points can be the main application file (e.g. `index.js`) that is defined in the `webpack.config.js` file as follow:

```
module.exports = {
  entry: './src/index.js',
  // Other configurations
};
```

Output Points: The output configuration specifies where Webpack writes the generated bundles and associated assets. It defines the filename pattern for the generated bundles, the directory where they will be placed, and any additional options like source maps for debugging. A typical output configuration might specify a filename like `bundle.js` and an outputPath like `dist` directory. The `output` section of the configuration file will look like this:

```
module.exports = {
  output: {
    filename: 'bundle.js',
    path: __dirname + '/dist',
  },
};
```

Loaders: Loaders are responsible for transforming various file types before they are included in the final bundle. Webpack doesn't understand file types like SCSS, TypeScript, or images directly. Loaders act as plugins that convert these files into a format Webpack can process, such as JavaScript or CSS. For example, a sass-loader can convert SCSS files into standard CSS, while a typescript-loader can compile TypeScript code into JavaScript. Loaders allow using modern features and frameworks in projects by enabling the use of different file formats. Loaders are defined in the `module.rules` section of the configuration:

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: 'babel-loader',
      },
      {
        test: /\.scss$/,
        use: ['style-loader', 'css-loader', 'sass-loader'],
      },
    ],
  },
};
```

Plugins: Plugins extend Webpack's functionality beyond basic bundling. They can perform various tasks like code optimization (minification, tree-shaking), asset management (copying fonts or images), code analysis (linting), and more. The Webpack ecosystem offers a vast array of plugins available for download, allowing you to customize the build process to suit your specific needs. Popular plugins include UglifyJSPlugin for minification, CopyWebpackPlugin for asset management, and Webpack-BundleAnalyzer for visualizing your bundle composition. Plugins are included in the `plugins` array within the configuration. Below is the example of UglifyJSPlugin that is used for minification:

```
const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
module.exports = {
  plugins: [
    new UglifyJSPlugin({
      test: /\.js$/, // Apply minification only to JavaScript files
      exclude: /node_modules/, // Exclude files from the node_modules
   directory
      uglifyOptions: {
        output: {
          comments: false, // Remove all comments
        },
      },
    }),
  ],
    // Other configurations...
};
```

Mode: This option allows to configure Webpack for different environments (e.g., development, production). The mode can be set to development for optimizations focused on faster development cycles, while production optimizes for smaller bundle sizes and faster loading times:

```
module.exports = {
  mode: process.env.NODE_ENV || 'development',  // Set mode based on
    NODE_ENV or default to development

};
```

Devtool: Controls how source maps are generated. Source maps help in debugging by mapping the transformed code back to the original source code. For example, `devtool:'inline-source-map'` is commonly used in development for better debugging. Other devtool settings, such as source-map, cheap-module-source-map, and eval-source-map, offer different balances of build speed and debugging effectiveness, allowing you to choose the most appropriate setting based on your specific needs and environment:

```
module.exports = {
 devtool: 'inline-source-map',  // This setting generates inline
    source maps for better debugging

};
```

4.3.2 Webpack Dev Server

The Webpack Dev Server [56, 55] is an integral part of the development workflow. It is an invaluable tool that streamlines the development process by offering features like local development server functionality and Hot Module Replacement (HMR). This section delves into setting up the Webpack Dev Server and explores the benefits of HMR:

Setting Up the Webpack Dev Server: The Webpack Dev Server is included as a dev dependency in most Webpack projects. To install it using npm or yarn:

```
npm install webpack-dev-server --save-dev
```

The next step,is to add the dev server configuration to the `webpack.config.js` file using the devServer property:

```
module.exports = {
  devServer: {
    contentBase: './dist', // Specify the directory containing your
   static assets
    port: 8080, // Set the port for the development server (default:
   8080)
    hot: true, // Enable Hot Module Replacement
  },
};
```

This configuration sets the content base to the `dist` directory, specifies the port number, and enables hot module replacement. With the configuration in place, you can start the Webpack Dev Server using a script in your package.json file:

```
module.exports = {
{
  "scripts":
   { "start": "webpack-dev-server"}
}
};
```

Hot Module Replacement (HMR): Hot Module Replacement (HMR) is a revolutionary feature for development using Webpack Dev Server. It enables you to witness modifications made to your code reflected in the browser practically instantaneously, eliminating the need for full page reloads. This significantly enhances development efficiency and reduces the time wasted waiting for builds to complete. Table 6, summarizes the benefits and limitations of this approach. HMR operates through a series of steps:

- **Initial Build:** Webpack constructs your application and injects the code into the browser alongside the HMR runtime.
- **Code Changes:** Upon modifying a source file, Webpack detects the alteration and re-bundles the impacted module.
- **HMR Runtime:** The HMR runtime residing within the browser receives the updated module from the server.
- **Module Update:** The runtime replaces the outdated module in the application with the new version, without a full page reload.
- **State Preservation (Optional):** In certain scenarios, HMR can preserve application state (e.g., form data) during the replacement process.

Aspect	Details
Benefits of HMR	• **Faster Development Iteration:** HMR eliminates the need for full page reloads, allowing you to see changes reflected almost instantly. This significantly accelerates your development cycle. • **Improved Debugging:** With HMR, you can directly observe the impact of code changes, making it easier to identify and rectify bugs. • **Enhanced Developer Experience:** The seamless integration of code changes with the running application fosters a more productive and enjoyable development workflow.
Limitations of HMR	• **Certain Changes Might Require Reloads:** HMR might not function flawlessly for all code modifications. Complex state updates or UI changes might still necessitate full page reloads. • **Potential Bugs:** In rare cases, HMR can introduce bugs if the module replacement process leads to unexpected behavior.

Table 6: Benefits and Limitations of Hot Module Replacement (HMR)

5 Using Webpack in a Microfrontend Setup

In the previous section, we introduced the fundamentals of Webpack, its core concepts and its configuration, we also provided the many advantages of using it in the development process. Let us consider the example of a web project where we have multiple JavaScript files and CSS stylesheets : using the standard way we need to include each script and stylesheet manually in the HTML, which results in a large number of HTTP requests and issues with dependency managements. Instead, if we use Webpack

we can bundle everything into a singe file and ensure that dependencies are resolved in the correct way. This is a very basic proof that shows the importance of such a framework even in the simplest projects. However, in this section, we will tackle a concrete example of the use of this versatile module bundler in the setup of microfrontend, as it is most known for its utility in this domain.

We will create a very simple microfrontend project using the Angular Framework. The project idea is the management of an online movie shop, where a logged in user can chose movies to add to his cart and then go the the summary section where he can see everything that he added and perform the purchase. For the sake of simplicity, we used a public open source API, from the Jikan API, which is used to retrieve data about anime movies from MyAnimeList, in order to generate a list of movies that the user can chose from.

5.1 Webpack 5 : Module Federation

Introduced in Webpack 5, module federation, that we will use in our example, is a feature that enables efficient code sharing across multiple independent projects at run time. It facilitates the creation of a system of micro frontends, where each individual frontend application can be developed and deployed independently, but still share code with other parts of the application [54]. Here are the main three concepts of this approach:

- **Containers**: Each micro-frontend is a seperate project and acts as a container, encapsulating its own code and dependencies.
- **Remote Containers**: Containers can dynamically load modules from other remote containers at runtime based on their exposed functionalities, optimizing resource usage.
- **Shared Modules**: Certain libraries or dependencies can be designated as shared between containers. Webpack ensures that only a single copy of these shared modules is included in the final application, reducing duplication.

5.2 Walkthrough: Implementing the Example

5.2.1 Structuring the project and creating the mf-components:

The very first step is choosing a multirepo or monorepo architecture (all in one repository or each project in a separate repository). In this example, we chose the monorepo approach. Then we will need a base workspace, that is not an application, where we will store our seperate projects (each project represents a micro-frontend component). This is done using the following command, that will create a workspace without the `src` folder:

```
ng new demo-microfrontend --create-application=false
```

Now we will need to decide and create our independent components based on the functionalities of the application, we will need three components : `mf-shell` which will be responsible for controlling, integrating and orchestrating the rest of the components, `mf-shopping` which is the one responsible for showing all movies , and `mf-payment` where the user can see what he chose, and perform the payment. This is done executing the following commands (we also chose the style=scss and the routing option=true while creating those projects):

```
ng g application mf-shell style=scss --routing=true
ng g application mf-shopping style=scss --routing=true
ng g application mf-payment style=scss --routing=true
```

Now we will need to create a project of the library type named `commons-lib`, this is gonna be the shared library between all the different projects, in order to pass data from a `mf-component` to another:

```
ng generate library commons-lib
```

Until now the structure is as shown in the figure 11: a folder projects where inside there is our three components and the common library, all inside a global workspace:

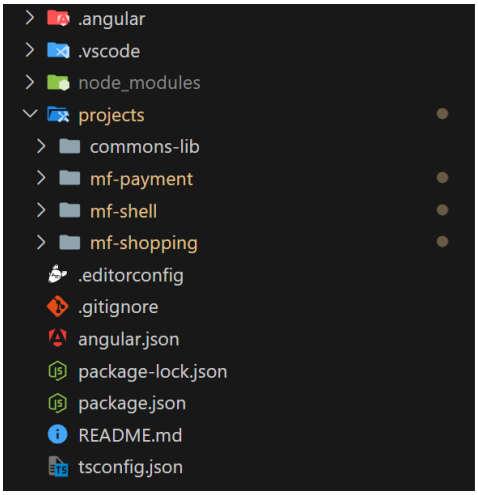

Figure 11: Microfrontend project stucture

5.2.2 Setup and configuration of module federation:

In order to use Webpack and module federation we need first to install an angular library called angular-architects and this is done running the following command:

```
npm i -D @angular-architects/module-federation
```

By examining the `package.json` file in the base workspace's `devDependencies` section, we can confirm that the line `"@angular-architects/module-federation":"14.3.14"` has been added, indicating successful installation. With this library now installed, we need to add module federation to each project individually. Additionally, we must define the port on which each project will run and specify whether it will be a host or remote type.

```
ng add @angular-architects/module-federation --project mf-shell
--port 4200 --type host

ng add @angular-architects/module-federation --project mf-shopping --
    port 4201 --type remote

ng add @angular-architects/module-federation --project mf-payment --
    port 4202 --type remote
```

This specifies that the `mf-shell` project will run on port 4200 and function as a host type (a type of microfrontend). This means that this in our example, we want the `mf-shell` project to render information and integrate other projects within itself. In other words, we will "embed" other projects inside this one. On the other hand, The `mf-shopping` and `mf-payment` projects run on ports 4201 and 4202, respectively, and are both of type remote. Executing this command creates, for each project, two files named `webpack.config.js` and `webpack.prod.config.js`, as shown in Figure 12. These files export a set of configurations, that defer based on the type. Lets take a look of the content of the `webpack.prod.config.js` file of the `mf-shell` project:

```
module.exports = withModuleFederationPlugin({
  remotes: {
    mfShopping: "http://localhost:4201/remoteEntry.js",
    mfPayment: "http://localhost:4202/remoteEntry.js",
  },
```

it includes a remotes option that indicates which other microfrontend components will be integrated into this project since it is the host. In our case, we will include `mf-shopping` and `mf-payment`, because at the end each project exposes in its compilation a file remoteEntry.js, which will be in charge of loading the project.

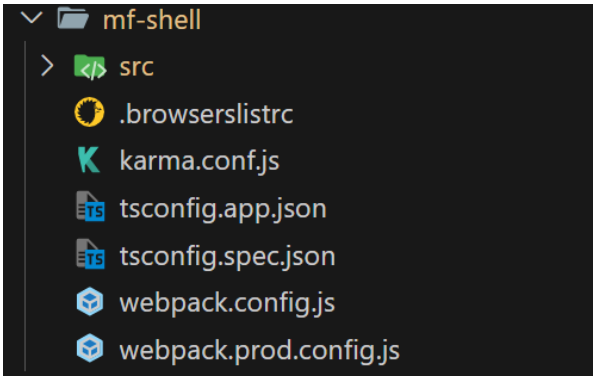

Figure 12: mf-shell

Conversely, the other two projects, being of the host type, do not contain the remotes configuration in their `webpack.config.js` files. Lets take a look at the content of this file for the `mf-Shopping` project:

```
module.exports = withModuleFederationPlugin({
  name: "mfShopping",
  exposes: {
    "./ProductsModule":
      "./projects/mf-shopping/src/app/products/products.module.ts",
  },
});
```

The `exposes` configuration in Webpack's Module Federation specifies the module or this project that is gonna be shared with other projects. In our case, the `ProductsModule` from the `mf-shopping` project is made available for other projects to import and use. The key (`"./ProductsModule"`) serves as the reference name for remote projects, while the value : (`"./projects/mf-shopping/src/app/products/products.module.ts"`) denotes the path to the actual module being shared.Inside this module all components and services of this projects are included. This means that we are sharing a module containing everything of this project. This facilitates dynamic importing and sharing of module functionality among independently developed microfrontends. The same is also done for the other component that will expose `./PaymentModule`.

5.2.3 Configuring the communication channel : commons-lib

In the majority of microfrontend projects, information exchange, including data transmission and reception, is often necessary. In this scenario, we require the transfer of the user-selected movie list from the `mf-shopping` project to the `mf-payment` project for visualization and payment processing. To achieve this, we can utilize observables such as Subject or BehaviorSubject, which are components within the Angular RxJS library. Adding these components to the `package.json` of the library project is essential. It's important to ensure compatibility by using the same version of RxJS as specified in the base project's `package.json` file. So in the `commons-lib` project we will install it using :

```
npm install rxjs
```

It's important to keep in mind that the library functions as a distinct project. To utilize this library, it must first be compiled, allowing it to be referenced from each microfrontend project. This can pose a minor challenge in monorepos. Fortunately, Angular has addressed this issue by providing a solution for referencing library-type projects.

To address this, adjustments need to be made within the `tsconfig.json` file of the base workspace project. By default, when the library is created, it's configured to search for the `commons-lib` library in the `dist/commons-lib` folder as shown below:

```
"compilerOptions": {
    "baseUrl": "./",
    "paths": {
      "@commons-lib": ["dist/commons-lib"],
    },
```

However, this only occurs post-compilation. During development, when adjustments and testing are required, a direct path to the library project can be defined, bypassing the need to reference the compiled version. Once confident in the functionality of the library, it can be compiled, and subsequent references can be made to the compiled version rather than the project itself.

In our example, the following modification can be made to refer to the project of the library, specifically targeting the elements intended for sharing:

```
"compilerOptions": {
    "baseUrl": "./",
    "paths": {
      "@commons-lib": ["projectsdist/commons-lib/src/public-api.ts"],
    },
```

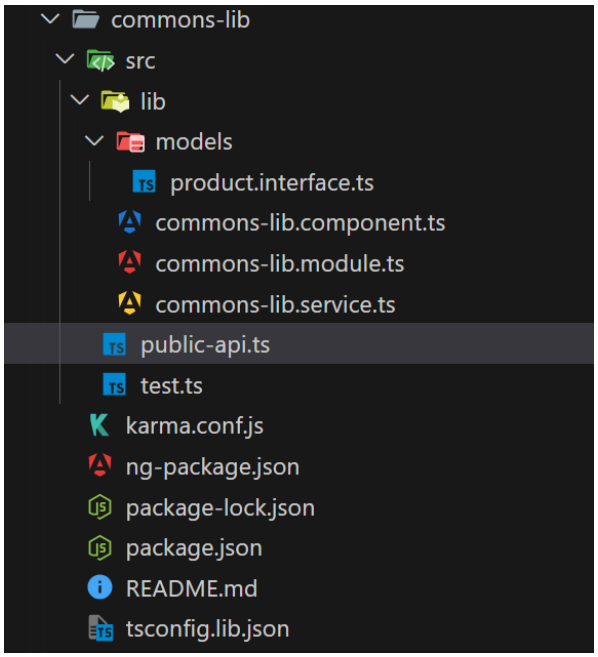

Figure 13: commons-lib

Here we are saying that we want to share the file `public-api.ts` of the `commons-lib` library. So only whats inside this file can be acessed by other projects. The content of our `pubic-api.ts` is the following:

```
export * from './lib/commons-lib.component';
export * from './lib/commons-lib.module';
export * from './lib/commons-lib.service';
export * from './lib/models/product.interface';
```

So basically we are exporting everything in this library. In our case we only used a service and interface but everything was exported to show that we can also create components, modules all inside the common library and then export everything and share it with all other projects.

Subsequently, the library must be added to each microfrontend project where its use is required. To incorporate the library within a microfrontend, a configuration needs to be added to the `webpack.config.js` file of each microfrontend project:

```
module.exports = withModuleFederationPlugin({
    shared: {
    ...shareAll({
      singleton: true,
      strictVersion: true,
      requiredVersion: "auto",
    }),
  },
  sharedMappings: ["@commons-lib"],
});
```

In the `sharedMapping` property, we list all aliases of external libraries intended for use in each microfrontend (in our case there is only one `commons-lib`). Angular architects will subsequently inject an instance of the library into the microfrontend, resulting in a cleaner interaction with the library. As for the `shareAll()`, it is a function that shares all dependencies from the remote projects. The options provided `(singleton: true, strictVersion: true, requiredVersion: "auto")` ensure that the shared dependencies are singletons, strict versions are enforced, and required versions are determined automatically.

5.2.4 Configuring Routes and the html of the shell component

Now we need to configure the `app-routing.module.ts` of the `mf-shell` component to embed inside it the other projects. Since we are working with modules, we can use `loadChildren` to import them. However, because the modules from the other projects are external to the `mf-shell` project, we must declare an import path. To do this, we need to add a `custom.d.ts` file within the `src` folder of the `mf-shell` project as shown in figure 14. This enables TypeScript to recognize external modules.

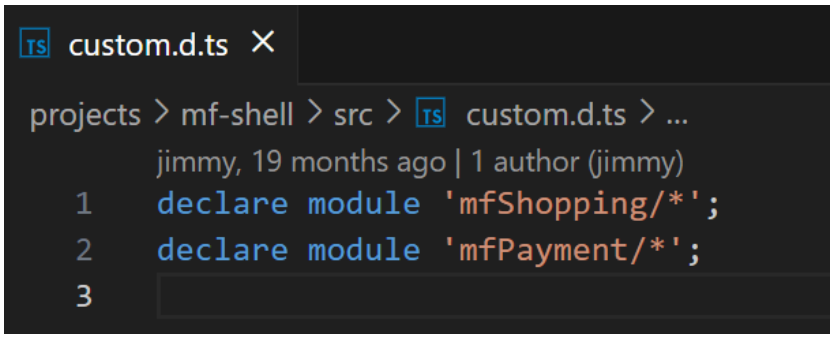

Figure 14: custom.d.ts file content in the mf-shell project

Consequently, we can import the `mfShopping` module in the app-routing module of the shell component, which has been declared and imported into the `custom.d.ts` file created earlier. Same for the `mfPayment` module. In the case of the payment path, as we can see in the figure 15, we used `loadComponent` instead of `loadChildren` because in our simple example , the `mf-payment` project has only one standalone component, that is a feature of angular that acts like a module. This was used to show that microfrontend projects can vary from small projects with one component (like in the `mf-payment` case) to more complex projects with modules that include more components (like the `mf-shopping` that has more components).

As shown in the `app.component.html` file in figure 16 of the `mf-shell` project, there is a section containing a div with a title and a router outlet. Depending on the path chosen from the available options in Figure 15, the appropriate microfrontend project will be rendered in place of the router outlet. This ensures that the title is always displayed along with either the payment or shopping project.

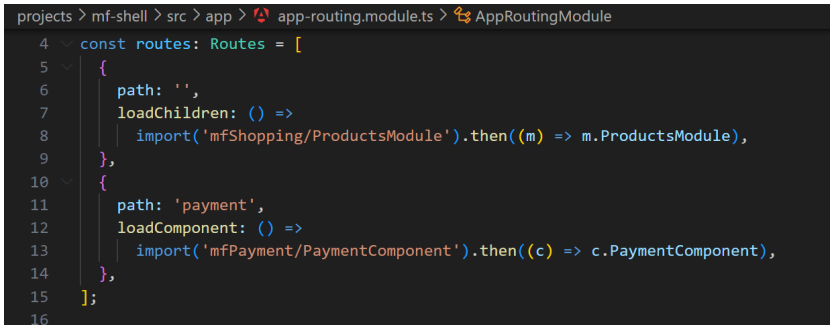

Figure 15: app-routing.module.ts file content in the mf-shell project

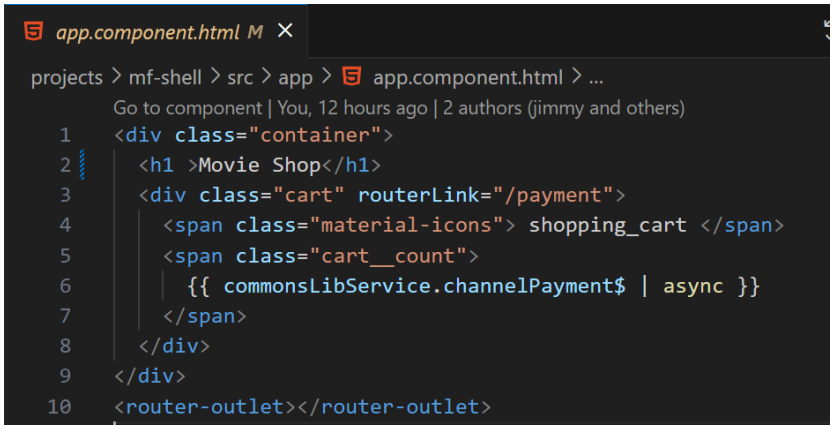

Figure 16: app.component.html file content in the mf-shell project

5.2.5 Configuring projects scripts

We can run each microfrontend project separately using `ng-serve` within each project (`mf-shell, mf-payment, and mf-shopping`). Alternatively, we can run all projects in parallel using a single command. To achieve this, we can install the `npm-run-all` library in the base project with the following command:

```
npm i npm-run-all
```

The next step, is creating a script for each project, and then create a script to run all projects in parallel, in the `package.json` of the base worskpace project, like this:

```
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test",
    "mf-shell": "ng s mf-shell",
    "mf-shopping": "ng s mf-shopping",
    "mf-payment": "ng s mf-payment",
    "all": "npm-run-all --parallel mf-shell mf-shopping mf-payment"
  },
```

After setting this up, we should be able to see the project running on the port specified for the `mf-shell` project (4200) by executing :

```
npm run all
```

5.2.6 Running and checking the behavior:

After executing the `npm run all` command, and opening the browser on port 4200 (that is the `mf-shell`) we can see the following, there is the title and the `mf-shopping` component and it is correct because as written in figure 15, it is the default empty path. As shown in figure 17, some css was used to highlight the sections of the projects, so we can see that the red border including everything that is the `mf-shell` host project and the green border show the `mf-shopping` project that is replacing the router outlet.

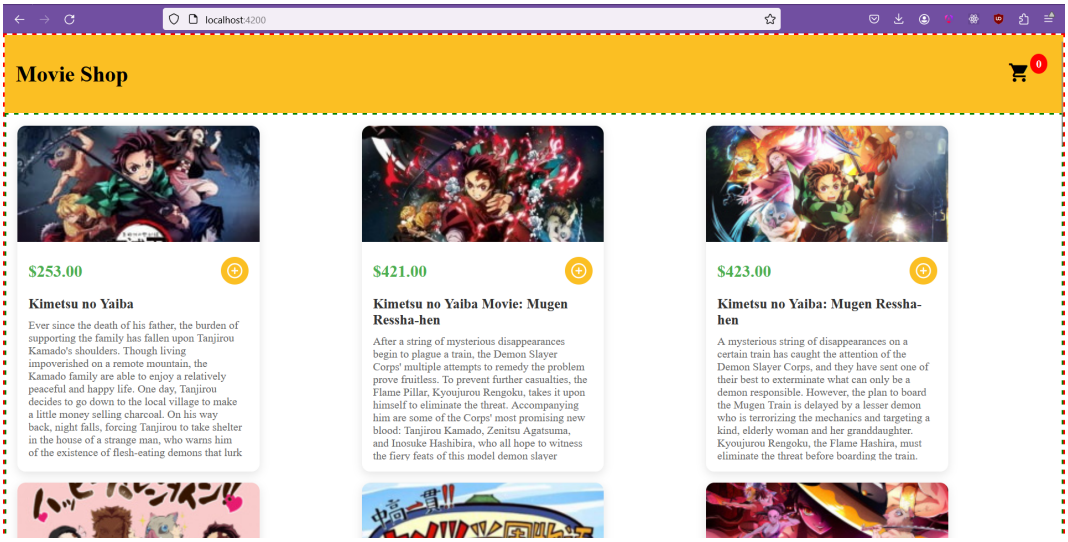

Figure 17: First page of the running microfrontend application

Now after the user has clicked on the + button of a bunch of movie cards that he would like to buy, and clicking on the cart logo that is next to the title, the page re-directs to the route `/payment` in figure 15, and loads the `mf-payment` project at the place of the router outlet as shown in the figure 18. In that way built a simple example where each component is a seperate project and they are sharing

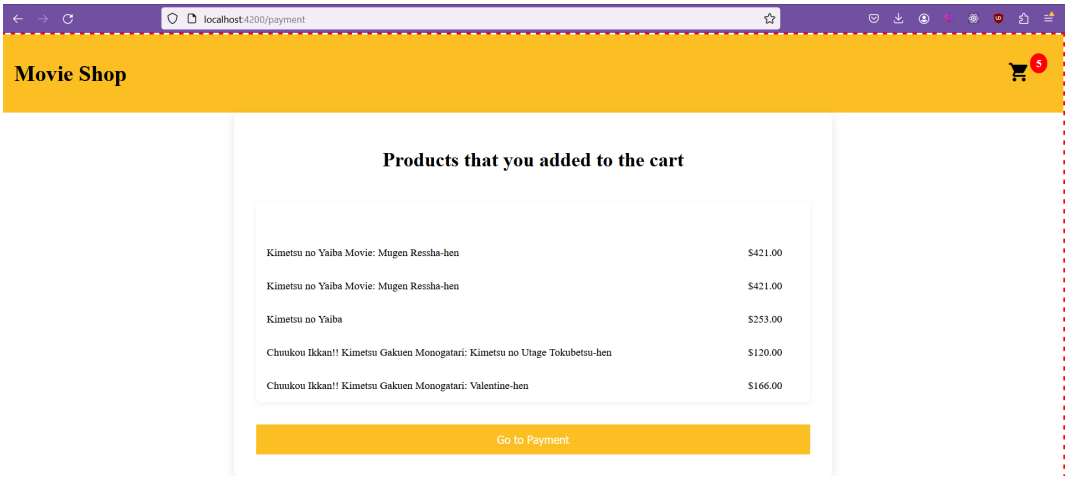

Figure 18: Second page of the running microfrontend application

informations between them sucessfuly (in this case they are sharing the movie list and the number of movies chosen that gets updated on the cart logo). This makes it simple to parallelize working on projects as one team can work on the list, another on the payment procedure for building powerful and more complex webistes. We can also check that if i go in my browser on the port 4201, i will be able to see the content of only the project running on this port , in this case the `mf-shopping` project, as shown below, the same for port 4202 for the payment project.

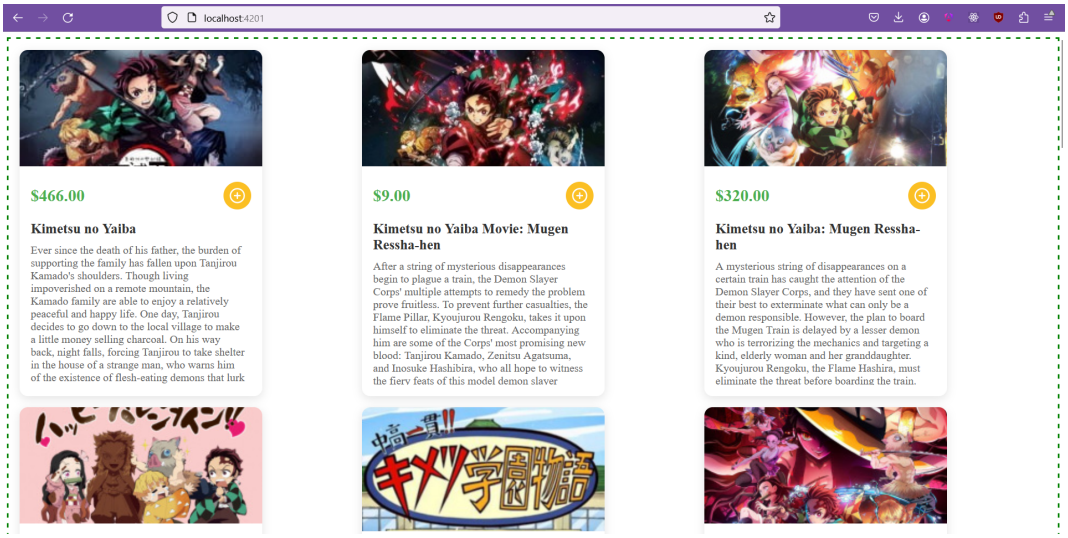

Figure 19: Running mf-shopping project

5.2.7 Details about the commons-lib:

Let's dive into the details of `commons-lib` to gain a better understanding of how data is shared between our projects. But first, let's remember that in the Angular framework, each angular component (such as a product card) comprises three files: an HTML file for the layout, a CSS file for the styling, and a TypeScript file for the logic and variables of the component, as shown in figure 20. Looking

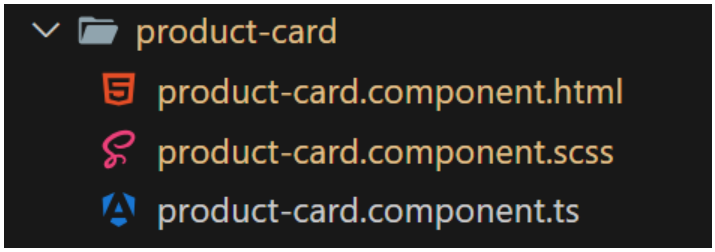

Figure 20: Files of angular component product-card

at the content of `product-card.component.ts`, we can see that in the constructor, an instance of `CommonsLibService` named `commonsLibService` was added to access the methods of this service. For example, in this case, the `sendData` method is used to add the name and price of the product each time the user clicks on the button of the product card.

```
export class ProductCardComponent {
  @Input() product?: IProductCard;

  constructor(private commonsLibService: CommonsLibService) {}

  clickCard(): void {
    this._commonsLibService.sendData({
      name: this.product!.name,
      price: this.product!.price,
    });
  }
}
```

Now going to the `commons-lib-service.ts` code shown below, we can see that when the `sendData` method is called, a variable `productList` of type `BehaviorSubject` is updated as well as a `channelPayment`

variable. A `BehaviorSubject` is a type of Observable in RxJS (Implements a Publish-Subscribe pattern) ,that we added before, that holds a current value and emits this value to new subscribers immediately upon subscription. It ensures that subscribers always receive the most recent value, as well as any subsequent values emitted by the subject. This makes `BehaviorSubject` useful for representing data that changes over time, such as application state or user inputs.

```
export class CommonsLibService {
  private _products: ICommonProduct[] = [];

  channelPayment = new BehaviorSubject<number>(0);
  productList: BehaviorSubject<ICommonProduct[]> = new
BehaviorSubject<ICommonProduct[]>([]);

  sendData(product: ICommonProduct): void {
    this._products.push(product);
    this.productList.next(this._products);
    this.channelPayment.next(this._products.length);
  }

}
```

Now going to the `payment.component.ts`, we can see that it also acesses the `commonLibService` in the constructor, and subscribes to the `BehaviorSubject` variable and updates a local variable of its own, and uses this local variable to show the list in the `payment.component.html`.Everytime the variable in the library is updated, so is the local variable in this component.

```
export class PaymentComponent implements OnInit {
  constructor(
    private commonLibService: CommonsLibService
  ) {}

  products: ICommonProduct[] = [];

  ngOnInit(): void {
    this.commonLibService.productList.subscribe({
      next: (res) => {
        this.products = res;
      }
    })
  }
}
```

This is how data is shared: one project accesses the `common-lib` and updates a variable of type `BehaviorSubject` in this library, while another project accesses the same library, takes the recent value of this variable by subscribing to this variable.

6 Software Testing

6.1 Definition and Significance of Testing

Testing in the context of software development is the process of evaluating a system or its components with the intent to find whether it satisfies the specified requirements or not. Testing involves the execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The main objective of testing is to identify errors, gaps, or missing requirements in contrast to the actual requirements of the project.Testing is essential in quality assurance because increased testing leads to higher product quality and fewer errors during real-world use.

Software testing has roots tracing back to the 1950s when it was an informal practice undertaken by developers themselves. However, as software complexity increased, the need for a more organized testing methodology became evident. Alan Turing introduced the concept of "program checking" in the 1940s [48], marking an early recognition of formal software testing. The emergence of software engineering in the 1970s further emphasized the importance of testing as a critical discipline. Glenford J. Myers' influential book, "The Art of Software Testing," published in 1979 [35], established fundamental principles that remain applicable today. Myers stressed the significance of testing not only for demonstrating software functionality but also for identifying errors to improve reliability and performance.

Software testing is indispensable for several reasons [46]. Firstly, it ensures the software's quality and functionality by systematically identifying and rectifying defects. This process aids in delivering a product that operates as anticipated across diverse conditions and scenarios, which is essential for ensuring user satisfaction and confidence. Secondly, an important domain where software testing is crucial is the security domain. In an world govenerd by prevalent cyber threats, testing helps find vulnerabilities that could be exploited by malicious entities. For example, security testing, a subset of software testing, concentrates on uncovering security weaknesses and ensuring that the software can safeguard data and maintain functionality even when under attack. Thirdly, software testing promotes cost efficiency. Identifying and resolving defects early in the development stage is considerably less costly than addressing issues after the software's deployment [39]. For example, a defect identified and corrected during the requirements phase may cost up to 100 times less to rectify than one discovered after the product has entered production. Additionally, software testing facilitates compliance with industry standards and regulations. Various sectors, including healthcare, finance, and aerospace, have stringent regulatory requirements for software. Lastly, software testing plays a critical role in continuous improvement and innovation. By providing insights into the software's performance and areas for enhancement, testing drives the iterative development process, leading to better and more innovative software products. After all software testing has a property once formulated by Dijkstra [11] as: "Program testing can be used to show the presence of bugs, but never to show their absence!".

So overall we can say that software testing is executing the software to [1]:(i) perform verification, (ii) to detect the mistakes and (iii) to achieve validation:

i. Verification: This process ensures that the software aligns with its specifications. [Are we building the product correctly?]

ii. Error Detection: Deliberate inputs are used to test the system's performance by inducing errors.

iii. Validation: This process confirms that the software meets customer expectations. [Are we building the correct product?]

Software testing is fundamentally a risk-centered activity. Figure 21 illustrates the relationship between testing costs and errors, revealing a sharp escalation in expenses. The overarching objective of testing is to strike a balance, conducting an optimal number of tests to minimize additional testing efforts. Thus, Figure 21 underlines the indispensable role of software testing in the world of software quality standards.

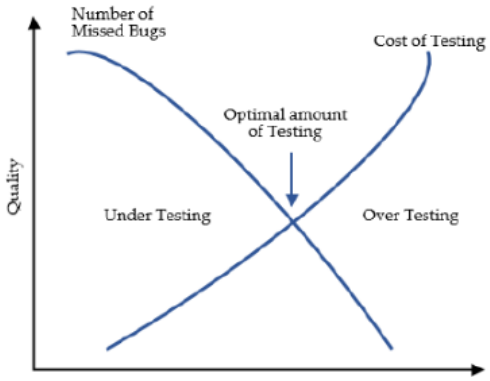

Figure 21: Ammount of testing

6.2 Software testing Methodologies

Software testing [42, 38, 21, 24] can be categorized into two main categories : static testing and dynamic testing. These techniques act as a team, each team reveals defects from a different side. Static testing analyzes code without running it, identifying potential issues early on. Dynamic testing, on the other hand, brings the code to life,by running it, uncovering problems that appear during execution:

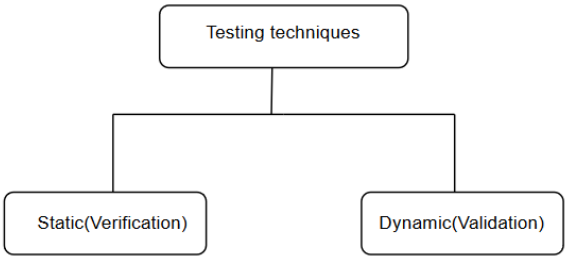

Figure 22: Testing techniques

6.2.1 Static Techniques

Static testing also knows as manual testing, is about testing the software without running the code. This method does not require highly skilled professionals since it does not include actual system execution. It is done in the initial phase of the Software Development Life Cycle (SDLC), and it is also known as verification testing. The aim of this technique is to identify and correct errors early in the development process. Static testing is conducted on documents such as Software Requirement Specifications (SRS), design documents, source code, test suites, and web page content, thus allowing for the evaluation of both code and documentation. Some of those static techniques include :

1. **Inspection**: This technique is primarily conducted to identify defects. Testers carry out the code step by step, using a checklist to methodically review the working document.
2. **Walkthrough**: This is an informal process led by the document's author. The author guides the participants through the document based on their thought process to achieve a common understanding. This technique is particularly useful for higher-level documents like requirement specifications.
3. **Technical Reviews**: A professional review process to ensure that the code aligns with technical specifications and standards. This may include reviewing test plans, test strategies, and test scripts to ensure compliance.

4. **Informal Reviews**: It is like an unofficial review of the document, and makes use of feedback recieved in order to introduce improvements.

6.2.2 Dynamic Techniques

In this technique, the code is executed. It requires the highly skilled professional with the proper domain knowledge. Dynamic testing involves testing the software for the input values, and output values are analyzed. Progressive testing is divided into two categories: White and Black box techniques.

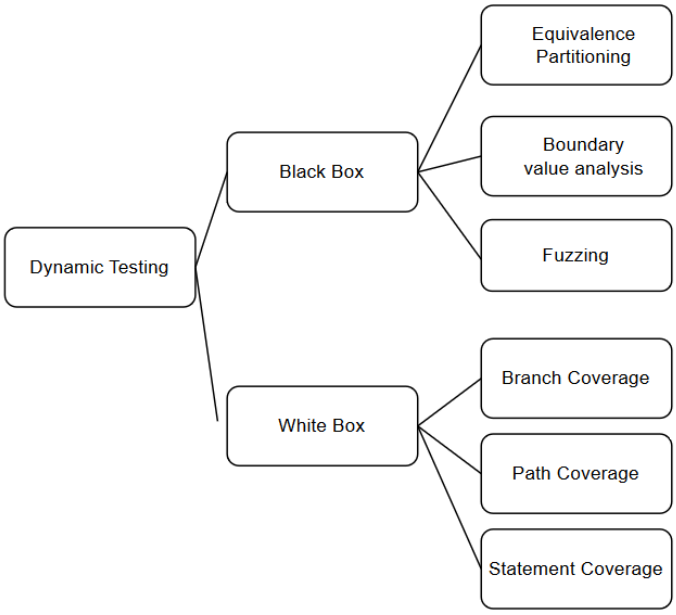

Figure 23: Dynamic testing techniques

White Box Testing: In white box testing , [4, 35], an engineer examines the software, using knowledge concerning the internal structure of the software. Hence, test data is collected and test cases are written using this knowledge, as shown in figure 24. It is reputated as an effective technique in detecting and resolving problems, because bugs will be found before they cause bickering. Some of the more widely known white box strategies are coverage testing techniques:

1. **Branch Coverage**: This technique tests each unique branch in a piece of code. For example, each possible branch at a decision point, such as a switch/case statement, is executed at least once, ensuring all reachable code is tested in that context.

2. **Path Coverage**: In contrast to branch coverage, path coverage involves testing complete paths within the code. This means every line of source code should be executed at least once during testing. However, achieving this is challenging for software with many lines of code. Consequently, engineers typically apply this technique in small, well-defined sub-domains that are critical to the software's functionality.

3. **Statement Coverage**: This technique aims to execute each statement in the software at least once. It has shown favorable results, making it quite popular due to previous validation.

Those kind of tests, provide a really good examination of the code, ensuring all paths are tested, which helps in detecting potential errors and vulnerabilities. They also aids in optimizing code by identifying hidden errors and eliminating dead code segments, ultimately leading to improved software quality and performance. Moreover, white-box testing facilitates the early detection of errors, reducing the cost and effort of fixing bugs later in the development process. However, white-box testing also

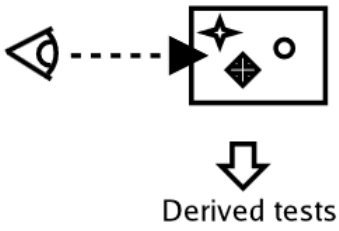

Figure 24: White box technique

comes with its set of disadvantages. One significant drawback is that it can be time-consuming and expensive due to the detailed level of testing required, especially for large and complex systems. Another challenge is the requirement for testers to possess deep knowledge of the internal workings of the software, making it less accessible for individuals with limited technical expertise. And it may not be suitable for systems with complex architectures, as the number of possible paths can be overwhelming.

Black Box Testing: Black box testing [3],treats a system like a sealed box. The tester focuses on what goes in (inputs) and what comes out (outputs) without worrying about the internal workings.They only have an understanding of what the software is supposed to do, not how it does it. This approach ensures the system behaves as documented, accepting valid inputs and producing expected outputs.Some of the most black box techniques are mentioned below :

1. **Equivalence Partitioning**: This method partitions the input domain of a program into equivalent classes, allowing for the derivation of test cases. By doing so, it effectively reduces the number of test cases required.

2. **Boundary Value Analysis**: This technique focuses on testing at boundaries, selecting extreme boundary values such as minimum, maximum, error, and typical values.

3. **Fuzzing**: This approach involves providing random input to the application, aiming to identify implementation bugs. It utilizes malformed or semi-malformed data injection in either automated or semi-automated sessions.

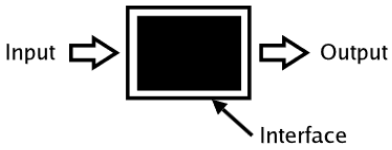

Figure 25: Black box technique

One significant advantage is their independence from the internal structure of the software, allowing testers to focus solely on the software's functionality from a user's perspective. They not require in-depth knowledge of the underlying code, making them accessible to a wider range of testers. However, black-box testing also has its limitations and disadvantages, the main one is that it provides an incomplete test coverage, as it relies on input/output behavior rather than comprehensive code analysis. This can lead to gaps in test coverage, potentially leaving critical parts of the software untested.

6.3 Software testing strategies

Software testing strategies [42, 43] offer a structured approach to incorporating various software test case design methods into a meticulously planned series of steps, thereby facilitating the successful development of software. Acting as a roadmap for testing, these strategies should be flexible enough

to accommodate customized testing approaches tailored to specific project requirements. Typically formulated by project managers, software engineers, and testing specialists, software testing strategies are under four primary types of testing that we will mention briefly for the sake of completeness:

1. **Unit Testing**: Involves testing individual units or components of the software in isolation to validate their functionality independently.In general, unit testing is categorized as a white-box testing technique since its primary focus is on assessing the code's implementation rather than simply verifying compliance with a predefined set of requirements.

2. **Integration Testing**: Integration testing serves as an effective method for constructing the program's structure and uncovering interface-related errors through testing. Its goal is to combine individually tested components and evaluate them as a cohesive group

3. **Acceptance/Validation Testing**:Acceptance testing is conducted to verify if a product aligns with predefined standards and user requirements. It involves external validation by the user or a third party to ensure the product meets specified criteria. Falling under the black-box testing approach, acceptance testing is characterized by minimal user involvement in the system's internal workings.

4. **System Testing**: Evaluates the software as a whole, testing its behavior and performance against specified requirements to ensure it meets quality standards and performs reliably.

6.4 Limitations of Manual Testing

Manual testing, while essential for certain aspects of software quality assurance, has several limitations that can impact the efficiency and effectiveness of the testing process. One of the primary drawbacks is the time-consuming nature of manually executing tests, particularly for large and complex applications. This can lead to longer development cycles and delayed releases. Additionally, manual testing is prone to human error, which can result in inconsistent test execution and overlooked defects. It also lacks the scalability needed for extensive regression testing, as manually re-running a large suite of tests for every code change is impractical. Furthermore, manual testing is less effective for performance and load testing, where automated tools can simulate thousands of concurrent users and capture precise metrics. These limitations highlight the need for integrating automated testing alongside manual efforts to achieve comprehensive and reliable software testing.

6.5 Automated Testing Frameworks

Automated testing [36, 47] frameworks are essential tools in modern software development, designed to enhance the efficiency, reliability, and scalability of the testing process. These frameworks provide a structured environment for creating, managing, and executing automated test scripts, allowing for consistent and repeatable test execution. By automating repetitive and time-consuming tasks, they free up valuable resources, enabling testers to focus on more complex and exploratory testing activities. Automated testing frameworks also facilitate continuous integration and continuous delivery (CI/CD) practices, ensuring that code changes are automatically tested and validated, thus accelerating the development cycle. Additionally, they offer robust reporting and logging capabilities, providing detailed insights into test results and helping to quickly identify and address defects. Popular frameworks like Selenium, JUnit, and TestNG support a wide range of testing needs, from functional and regression testing to performance and security testing, making them indispensable for maintaining high software quality in fast-paced development environment. Table 7, shows some of the most popular frameworks and their key features. In the next part of this thesis, we will focus on Cyperess and its capabilities in the context of automated browser ennd to end testing for web applications, and we will dig deeper into what this framework has to offer as well its challenges and limitations.

Framework	Description	Key Features
Selenium	Open-source web automation tool that supports multiple programming languages	Cross-browser testing Supports various browsers (Chrome, Firefox, Safari, Edge) Integration with popular programming languages (Java, Python, C sharp) Extensive community support
Appium	Open-source mobile automation framework for native, hybrid, and mobile web apps	Cross-platform testing (iOS, Android) Supports multiple programming languages (Java, Python, JavaScript) No need for app source code Seamless integration with Selenium
TestNG	Testing framework inspired by JUnit and NUnit, with added features	Flexible test configuration (e.g., test prioritization, dependency testing) Support for data-driven testing Built-in parallel test execution Extensive reporting capabilities
JUnit	Standard unit testing framework for Java applications	Simple annotation-based testing Easy integration with build tools (e.g., Maven, Gradle) Parameterized tests Support for test suites and categories
Cypress	Fast, easy and reliable testing for anything that runs in a browser	Built-in support for modern web development technologies (e.g., React, Vue, Angular) Real-time DOM manipulation and automatic waiting Time-traveling for debugging Automatic screenshots and video recording

Table 7: Comparison of Automated Testing Frameworks

7 Leveraging Cypress for Robust Web Application Testing

7.1 Overview of Cypress

7.1.1 What is Cypress?

Cypress [13, 49] is an open-source end-to-end testing framework that enables developers to write tests in JavaScript. It provides a robust and user-friendly platform for writing, executing, and debugging tests. Unlike traditional testing tools, Cypress is designed to tackle the challenges of testing dynamic and complex web applications, making it a favored choice among developers.

By operating directly in the browser, Cypress allows real-time interaction with the application as it runs. This unique approach offers numerous advantages, including increased speed, simplified setup, and enhanced visual debugging capabilities, all of which lead to a more efficient and effective testing process.

The framework provides a powerful suite of features that facilitate test creation and execution, such as automatic waiting, time-travel debugging, and detailed error messages. These features enable developers to quickly identify and resolve issues, thereby improving the overall quality and reliability of their applications.

7.1.2 Advantages of Cypress

Cypress provide many advantages [23] to the world of automation web testing, some of the main ones:

Increased Speed:One of the primary advantages of Cypress is its speed. Traditional testing frameworks often experience performance issues due to the need to manage and synchronize multiple components. Cypress, however, operates directly in the browser, eliminating these bottlenecks and significantly reducing test execution time. This increased speed allows for more frequent and comprehensive testing, enabling developers to identify and resolve issues earlier in the development cycle.

Simplified Setup: Setting up Cypress is a much easier compared to other testing tools. No need to mess with a bunch of configurations or manage dependencies. Just one command, and we're ready to start writing tests. This makes it simple for anyone to learn and use Cypress, and lets teams integrate it quickly into their workflow. Plus, Cypress supports different types of testing, like end-to-end, integration, and unit testing, all in one place. This means there is no need for a bunch of different tools, simplifying the test suite management. With Cypress, we have everything we need in a single package.

Visual Debugging: Traditional testing tools can be hard to understand when a test fails. We might have to sift through logs and code to figure out what went wrong. Cypress does things differently, offering visual debugging that helps us see the problem clearly.

When a test fails, Cypress takes a snapshot of the app at that exact moment. This snapshot acts like a picture, showing you exactly what the app was doing when the test failed. Cypress lets you rewind the app's state within the test, like a time machine, allowing us to see what happened leading up to the failure, step by step. This amazing visual insight makes it super easy to diagnose and fix problems in shorter time periods.

7.2 Cypress: Simplifying End-to-End Testing with Core Concepts

Cypress changed the way in which testers or developers view end-to-end (E2E) testing for web applications. Cypress differs from traditional frameworks because it doesn't need meticulous control over every step, instead Cypress embraces a more streamlined approach centered on its core concepts that improve overall testing efficiency. Let's take a deep dive into four of these essential features [34, 14]:

7.2.1 Declarative Syntax: Focus on What, Not How

One of the most notable advantages of Cypress is its declarative syntax. Unlike the traditional imperative style of testing, where you specify each action step-by-step (e.g., click here, then type there), Cypress allows you to describe the final desired state of your application. It then automatically determines and performs the necessary actions to reach that state.

Imagine a case where you want to test a login button in an authentication page. In Cypress, your test would look something like this:

```
cy.get('#username').type('testuser');
cy.get('#password').type('password123');
cy.get('button[type="submit"]').click();
```

This code clearly defines in a very easy and intuitive way what the tester is trying to do: the username field should be populated with the value "testuser", the password field should be filled with "password123" and the submit button should be clicked. The framework handles converting these desired states into the appropriate actions, such as finding elements, executing necessary interactions, and waiting for the page to transition as needed.

This core concepts introduces several benefits such as :

- **Improved Readability:** Tests become more intuitive and easier to understand by focusing on the intended outcome rather than how it is done.
- **Reduced Boilerplate Code:** Repetitive code blocks that handle element identification and low-level actions are eliminated, making the life of the tester much easier because it significantly reduces test size and complexity.

- **Enhanced Maintainability:** When the UI structure changes, because of modifications the only thing to change in the tests is the desired state descriptions .Cypress automatically adapts its internal actions to accommodate the UI updates.

7.2.2 Automatic Waiting: No More Explicit Waits

A major time-saving feature of Cypress is its automatic waiting capability. In other testing frameworks it is required to manually insert wait commands to ensure elements are ready before interacting with them, Cypress handles this automatically. It uses various strategies to wait for elements to become visible, interactive, and for network requests and animations to complete. This removes the need for manual wait commands, resulting in:

- **Concise Test Code:** Tests are no longer cluttered with repetitive wait statements, improving readability and reducing overall code size. For example, instead of writing code to wait for an element to appear, you simply describe the final state, and Cypress ensures the element is ready before proceeding.
- **Increased Reliability:** Tests are less prone to failures due to timing issues or unexpected delays, resulting in more stable and reliable test execution. For instance, if an element takes longer to load than usual, Cypress will automatically wait until it is ready, preventing potential test failures.
- **Simplified Maintenance:** As the application evolves, there is no need to constantly modify wait times based on potential UI changes. Cypress automatically adapts its waiting logic, ensuring tests remain robust without requiring frequent updates. This means if a new animation is added to the page, Cypress will handle the wait without any manual intervention.

7.2.3 Time Travel (Experimental): Exploring the Future

Cypress offers an interesting experimental feature known as time travel. This allows the developer to manipulate time within inside tests, enabling the simulation of scenarios involving time-sensitive actions or data. Lets take the example of testing a functionality that becomes available only after a specific time period. With time travel, we are able to programmatically move forward in time to trigger the desired behavior.This feature can be particularly useful for testing scenarios like: a countdown timer that unlocks a feature after a specific duration, or any time-based conditions that need to be validated within the application. Here's an overview of how we can achieve this behavior:

```
cy.clock(); // Enables time manipulation
cy.get('#time-sensitive-action').click();
cy.tick(30000); // Moves time forward by 30 seconds
// Assert on the expected outcome after the time jump
```

7.2.4 Real-Time Reloads: Streamlined Development Workflow

Cypress provides real-time reloading during the testing sessions. Any changes made to the application code are automatically reflected in the running tests. This eliminates the need to manually restart the tests after every code modification, allowing to witness and monitor the immediate impact of the changes on the test suite. This leads to :

- **Faster Development Cycles:** Developers can iterate on the code and observe the corresponding test results almost instantaneously, leading to quicker feedback loops.
- **Improved Efficiency and Productivity:** No more wasting time restarting tests after making code changes, as the framework allows focusing more on writing code and less on managing the testing process.

By embracing these core concepts, Cypress encourages testers to write cleaner, more maintainable, and efficient end-to-end tests for web applications. It simplifies the testing process, allowing them to focus on what truly matters.

7.3 Integrating Cypress with CI/CD Pipelines

Automated testing has become a crucial part of modern software development practices, enabling teams to catch bugs early and maintain code quality throughout the development lifecycle. Integrating Cypress into the CI/CD pipelines [12] brings automation to the forefront, enabling developers to run tests automatically with every code change. In that way we ensure that new features and bug fixes are thoroughly validated before deployment, reducing the risk of introducing regressions into the production environment. Here are some key benefits of CI/CD integration:

- **Faster Feedback Loops**: With automated testing, developers receive immediate feedback on the impact of their code changes. Failed Cypress tests highlight potential issues, forcing developers to resolve them before merging their code into the main branch.
- **Improved Code Quality**: Incorporating rigorous testing standards as part of the CI/CD process helps maintain higher code quality. Cypress tests act as a safety net, ensuring that new features and changes do not break existing functionalities.
- **Streamlined Deployment Process**: Continuous Delivery pipelines automate the deployment process, enabling teams to release updates to production more frequently and with greater confidence. Cypress tests ensure that each deployment meets the desired quality and functionality criteria.

7.4 Simple test example with Cypress

7.4.1 Installation and Setup

First we need to open a terminal in the project directory and run `npm init` to initialize a new `package.json` file if we don't already have one. Then, we install Cypress by running:

```
npm install cypress --save-dev
```

After the installation completes, to open Cypress for the first time we can run:

```
npx cypress open
```

This command will launch the Cypress Test Runner and automatically create the necessary folder structure (the `cypress` folder) along with example test files. Then we can now start writing tests in the `cypress/integration` directory and configure the Cypress settings in the `cypress.json` file.

7.4.2 Explanation of the Cypress Test Suite Example

This example (full code shown in appendix A) showcases a Cypress test suite designed to demonstrate testing functionalities for a to-do list application. In this section we will break down some of the code, block by block and detail its meaning and functionalities:

Reference: This line is a type definition for TypeScript to recognize Cypress commands.

```
<reference types="cypress" />
```

Test Suite Structure: The test suite is wrapped in a `describe` block with the title "example to-do app". This block groups all related tests under a single descriptive name. Inside the `describe` block, multiple `it` blocks define individual tests, each focusing on a specific functionality of the to-do list app.

```
describe('example to-do app', () => {
  // ... test definitions here ...
});
```

Before Each Hook: A `beforeEach` hook ensures that every test starts with a clean slate. Inside the `beforeEach` hook, the `cy.visit('https://example.cypress.io/todo')` command visits the to-do list application's URL before each test runs.

```
beforeEach(() => {
  cy.visit('https://example.cypress.io/todo');
});
```

Testing Default Items: The first test, "displays two todo items by default," verifies that the application starts with two pre-populated to-do items.

- `cy.get('.todo-list li')` retrieves all list items using the CSS selector `.todo-list li`.
- `should('have.length', 2)` asserts that the retrieved list items have a length of 2, indicating two default items.
- Additional assertions check the content of the first and last items using:

  `first().should('have.text', 'Pay electric bill')` and

  `last().should('have.text', 'Walk the dog')`.

```
it('displays two todo items by default', () => {
  cy.get('.todo-list li').should('have.length', 2);
  cy.get('.todo-list li').first().should('have.text', 'Pay electric
   bill');
  cy.get('.todo-list li').last().should('have.text', 'Walk the dog');
});
```

Adding New Items: The "can add new todo items" test demonstrates adding a new item to the list.

- A variable `newItem` stores the text of the new item ("Feed the cat").
- `cy.get('[data-test=new-todo]')` on line 3, selects the input field using the data-test attribute and types the new item text followed by the enter key to submit it.
- Assertions verify that the new item was added:
  - `.should('have.length', 3)` checks if the total list items are now 3 (including the new item).
  - `.last().should('have.text', newItem)` confirms that the last item (the new one) has the expected text.

```
it('can add new todo items', () => {
  const newItem = 'Feed the cat';
  cy.get('[data-test=new-todo]').type(`${newItem}{enter}`);
  cy.get('.todo-list li').should('have.length', 3).last().should('have
   .text', newItem);
});
```

Checking Off Items: The "can check off an item as completed" test demonstrates marking an existing item as completed. `cy.contains('Pay electric bill').parent().find('input[type=checkbox]').check()` finds the element containing the text "Pay electric bill" and checks the corresponding checkbox. Assertions verify that the list item is marked as completed: `parents('li').should('have.class', 'completed')` asserts that the `li` element has the "completed" class applied.

```
it('can check off an item as completed', () => {
  cy.contains('Pay electric bill')
    .parent()
    .find('input[type=checkbox]')
    .check()
```

This code sample, shows how powerful this testing framework is, and how easy it is to describe tests within test suits and implement the behavior of the end user by exploring all the possible senarios and ensure the correct behavior of the application in each senario.

7.4.3 Running Tests in Cypress Test Runner

After setting up Cypress in the project, we can run the tests by executing the following command :

```
npx cypress open
```

This command launches the Cypress Test Runner, an interactive UI where we can see a list of the test files. We then select the test file to run the tests, and the Cypress Test Runner will display a browser window, executing the tests while showing real-time results.

Each test step is visually represented, allowing us to see exactly what happened at each stage. The runner also provides detailed logs and screenshots, making it easy to debug any issues that arise. All those features ensure that tests run reliably and provide valuable feedback to maintain the quality of the application.

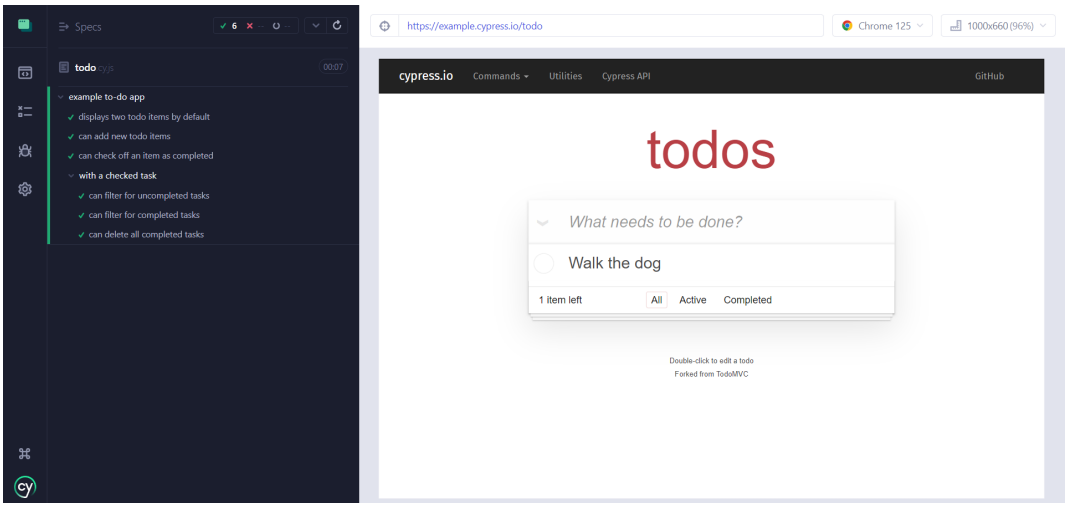

Figure 26: Cypress Test Runner

7.4.4 Cypress custom comands

Custom commands in Cypress empower developers to encapsulate frequently executed sequences of actions into reusable functions, thereby enhancing the clarity and manageability of tests. Through custom commands, routine tasks like logging in can be simplified into direct function calls. For example, consider the following custom command to log in to a website:

```
Cypress.Commands.add('login', (username, password) => {
  cy.visit('/login');
  cy.get('#username').type(username);
  cy.get('#password').type(password);
  cy.get('#login-button').click();
});
```

With this custom command, logging in becomes a single line of code:

```
cy.login('user123', 'password123');
```

7.5 Best Practices for effective Cypress Testing

Let us explore some key practices [8] to maximize the effectiveness of this framework:

- **Organizing Test Suites and Naming Conventions:** It is recommended to have an organized structure using folders or suites [14] that reflect functionality (e.g. login, search, checkout). Furthermore, employing clear and consistent naming conventions like (e.g. `test_login_successful.js`). Those two tipos enhance the readability and maintainability of the test suite.
- **Handling Asynchronous Operations:** Modern web applications heavily rely on asynchronous operations like API calls and DOM manipulations.Cypress provides mechanisms like waits and custom commands to handle these effectively. Utilizing '`cy.wait()`' carefully ensures that elements are loaded before interaction, and consider custom commands to encapsulate handling of common asynchronous actions within the application.
- **Implementing Data-Driven Testing:** To enhance test reusability and coverage, it is recommended applying data-driven testing. By leveraging fixtures (external JSON or JavaScript files) to store various test data sets, we can parameterize the tests to consume this data. This approach allows to execute the same test logic with different data combinations, improving the overall robustness of tests.
- **Leveraging Page Object Model (POM) Pattern:** The POM pattern promotes separation of concerns by isolating UI element interactions within dedicated page object classes. These classes encapsulate element locators and actions specific to a particular page (e.g., login page, product page). This improves code organization, maintainability, and reduces the risk of breaking tests due to UI changes.

By adhering to these best practices, we can leverage Cypress's capabilities to write effective, maintainable, and scalable automated tests that significantly contribute to the application's quality assurance process.

7.6 Challenges and Limitations of the Cypress Framework

While Cypress shines as a powerful tool for automated testing, it's crucial to recognize its limitations [34] to ensure a well-rounded testing approach. The framework prioritizes modern browsers, potentially leaving compatibility gaps for older ones. Its strength lies in End-to-End (E2E) testing, making it less ideal for scenarios requiring more detail control, such as unit or integration testing. Debugging intricate test failures can also be more challenging within a running browser compared to dedicated testing frameworks. Also Cypress commands run exclusively inside a browser, there is the lack of support for multiple browser tabs, and the inability to drive two browsers simultaneously. However, by acknowledging these limitations, we can strategically leverage Cypress within the testing suite, maximizing its effectiveness for E2E testing while potentially combining it with other tools if needed.

8 Practical Company use case scenario

8.1 Overview

The aim of this section is to present a practical company use case of transforming an application from a single-project, Angular Single Page Application (SPA) to a microfrontend architecture utilizing Webpack. This section will explore first, the content of the SPA project, that is the initial developed application and its structure. Then we will cover the analysis of transforming this project to a microfrontend approach, the detailed implementation of the new architecture, and the benefits achieved from this transition. In addition, the section will discuss the test suites that have been made using Cypress for each microfrontend project, highlighting the testing strategies and outcomes.

8.2 Initial Application Development

8.2.1 Technology Stack

The initial application that i developed for Wave Informatica SRL utilized Angular as the primary technology stack, adopting the Single Page Application (SPA) framework due to its robust capabilities in building dynamic and responsive web applications. Angular, a widely-used front-end framework, offers a comprehensive suite of tools and features, including powerful two-way data binding, dependency injection, and a modular component-based architecture. The application was structured as a single project that encompassed all modules, components, and services, resulting in a monolithic frontend architecture. This approach, was initially chosen for its simplicity and ease of development.

8.2.2 Functionality and features

The application has a set of features a little similar to a drive. It featured comprehensive folder management capabilities, allowing users to navigate through folders, upload, download, and delete files at a certain level. Additionally, the application supported advanced functionality such as visualizing Excel files directly within its interface, enhancing user interaction and productivity. Key functionalities included:

- **User Authentication:**This login page lets users enter their username and password to access the application, with the option to choose their preferred language.

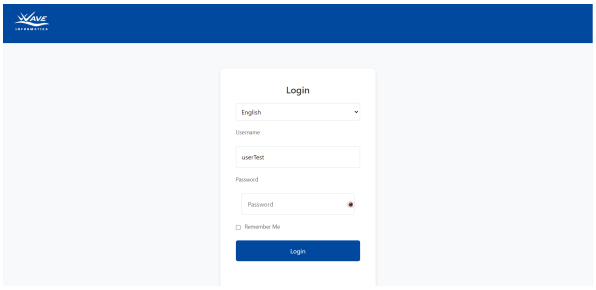

Figure 27: SPA Login page

- **Tree View Navigation:** A hierarchical tree view displaying all folders, enabling users to click on each folder to view its contents.
- **Dynamic Content Table:** A table located to the right of the tree view that dynamically updates its content based on the selected folder.
- **Search Bar:** A comprehensive search functionality with filters to quickly find specific items within the table.

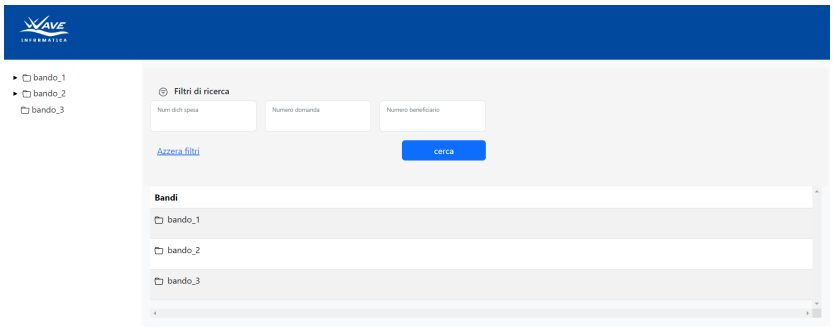

Figure 28: TreeView-Table-Filter

- **File Management:** The ability to upload, download, and delete files within selected folders. Users could manage files efficiently with modals for confirmation of deleting and uploading files.

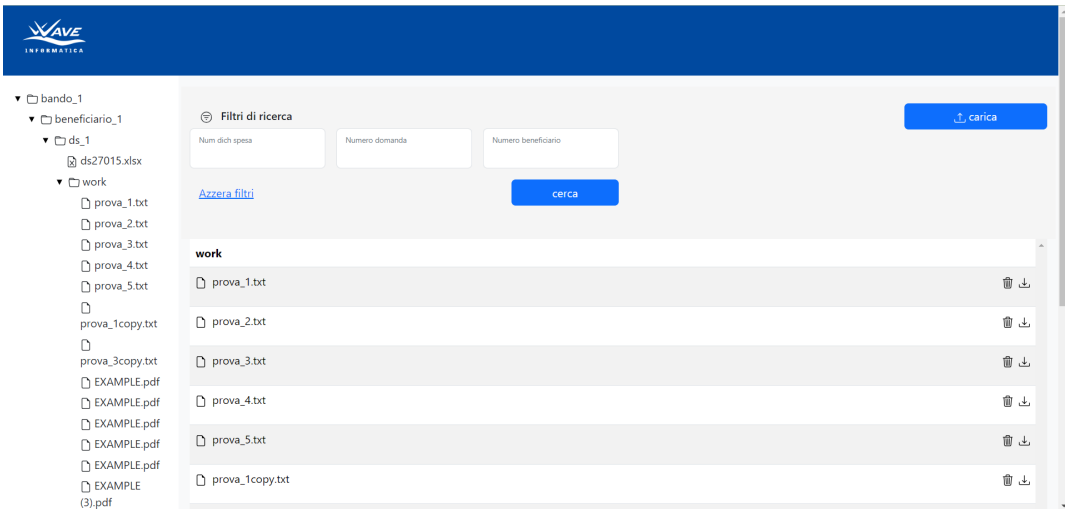

Figure 29: Upload-Download-Delete functionalities

Figure 30: Conferma Modal

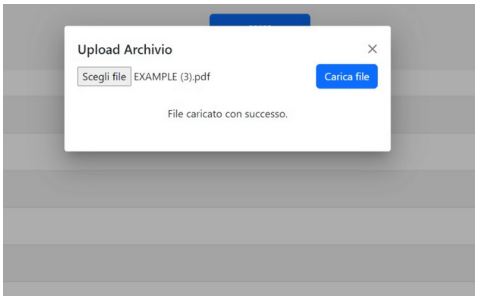

Figure 31: Upload Modal

- **Excel Visualization:** Clicking on an Excel file in the tree view allows users to visualize its content in the table area, replacing the standard table view with a spreadsheet-like interface for better data comprehension.
- **Backward navigation and versioning:** By selecting a version number from the menu on the right, users can choose different versions of the Excel file, each containing distinct data. They can navigate back to the table view by clicking the backward button.

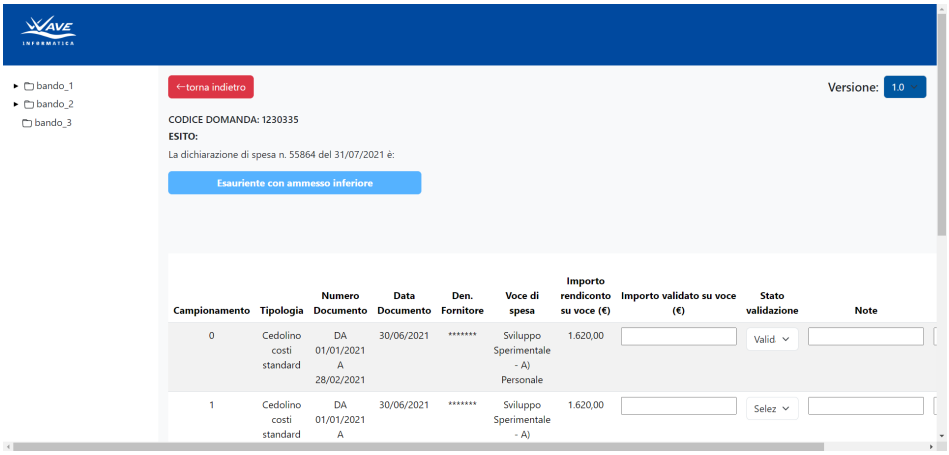

Figure 32: SpreadSheet -Navigation-Versions

8.2.3 Application Architecture

Let us have a look at the architecture of this application, that was designed as a single project containing all essential modules, components, and services. The application included:

- **Modules:** Modules in Angular are logical groupings that help organize an application into cohesive blocks of functionality. They consist of components, directives and services related to a specific feature or aspect of the application. Modules encapsulate code, making it easier to manage and reuse across different parts of the application. The application is organized into four main modules shown in figure 33:
  - Authentication Module: that includes all the login forms and services required for authentication.
  - Shared Module: that includes all shared services, shared components such as the confirmation modal, the sucess modal and components used many times across the application.
  - Treeview Module: that includes the tree-view component and service to get the data for the treeview.
  - Workspace Module: that includes the table component, spread-sheet component and their related services.

  Each module contains a dedicated `component` folder housing UI components specific to its functionalities, all declared and managed within their respective `module.ts` files (e.g. `workspace.module.ts`). Modules with multiple views or routes also include a `routing.module.ts` file (e.g. `workspace-routing.module.ts`) to define navigation paths and manage routing configurations.

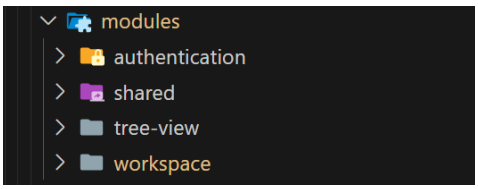

Figure 33: Modules in SPA

- **Components:** Components are the basic building blocks of Angular applications. They are reusable UI elements that encapsulate the HTML, CSS, and logic for a part of the user interface. Each component controls a part of the screen, with its own view and behavior, making it easier to develop and maintain complex user interfaces. Here is an figure showing the components of the workspace module, in this case we have three components:

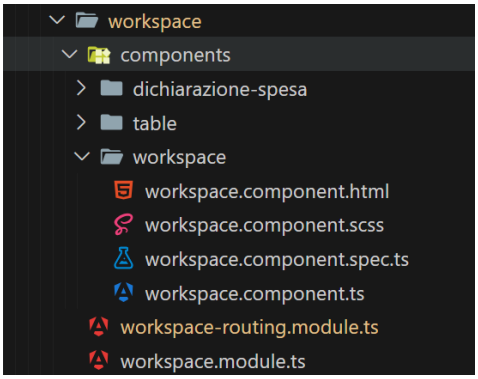

Figure 34: Components of the workspace module

- **Services:** Services in Angular are singleton objects that are instantiated once and shared across the application. They are used to encapsulate reusable data and business logic that can be accessed and used by different parts of the application, such as components, directives, and other services. Here are some services that we have in our shared module:

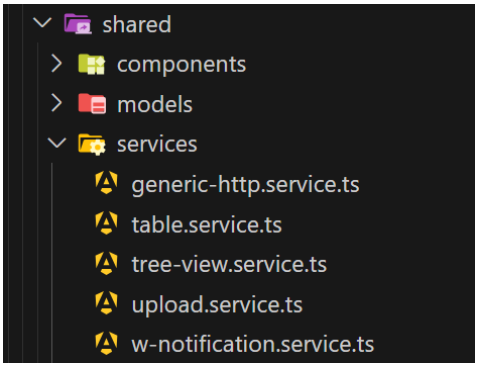

Figure 35: Servuces of the shared module

- **Routing:** Angular Router provides a powerful way to define navigation paths, map them to components and handle navigation events. It allows users to navigate between different parts of the application without reloading the entire page, providing a seamless and responsive user experience. Routing also supports features like route guards, lazy loading, and parameterized routes. Here are some examples of routes in the `app-routing.module.ts` and the `workspace-routing.module.ts` respectively, mapping each route to its corresponding module or component:

```
const routes: Routes = [
  { path: '', redirectTo: 'login',pathMatch: 'full'},
  { path: 'workspace', loadChildren:() => import('./modules/workspace/
    workspace.module').then(m=>m.WorkspaceModule) },
  { path: 'login', loadChildren:() => import('./modules/authentication
    /authentication.module').then(m=>m.AuthenticationModule) },
];
```

```
const routes: Routes = [
  { path: '', component: WorkspaceComponent, children: [
    {path: '', redirectTo: 'table',pathMatch: 'full'},
    { path: 'table', component: TableComponent },
    { path: 'ds/:id', component: DichiarazioneSpesaComponent }, ] },

];
```

8.3 Implementation of Microfrontend Architecture

8.3.1 Technology Stack

The objective was to rebuild the existing functional application using the microfrontend approach, detailed in section 5, to leverage the advantages of this modern web methodology and compare it with the traditional SPA approach. This transformation was achieved using the following technologies:

- **Angular:** The Angular framework was utilized to develop each individual microfrontend project. Additionally, with this framework a base project (not an application) was created to host all microfrontends.
- **Webpack:** Used for module bundling and facilitating the microfrontend architecture.
- **commons-lib:** A shared library named `commons-lib`, as previously explained, was used for communication between the microfrontend components. Additionally, all dependencies needed across all projects were downloaded once in this library and shared among the other projects.

8.3.2 Microfrontend Structure

The initial step involved analyzing the SPA to identify which parts of the application could be segmented into independent microfrontends and determining how these microfrontends would communicate with each other. Common strategies for communication included shared services, custom events, or a global state management solution. Our choice was a the `commons-lib` library.

Given that our original application was divided into four modules, this served as a logical starting point for defining our microfrontends.

It was essential to have a `mf-shell` project as a host, similar to the structure used in our simple movie example. To decide on the remote microfrontends that would be integrated into the shell, it was clear that a `mf-login` project was necessary to handle all aspects of authorization and authentication. Thus, everything from the authentication module in the SPA was incorporated into this project.

Additionally, a `mf-dich-spesa` project was created because it contained the spreadsheet-like component and the separate services required to retrieve the data for this component. For similar reasons, a `mf-table` project was established.

Since the treeview and navbar needed to be present with other components, they were included in the `mf-shell` project. This ensured that these critical navigation elements were consistently available throughout the application. So after running the necessary commands to install webpack and creating the base, and running those following three commands to create the projects and the library, we ended up with the architecture in figure 36:

```
ng add @angular-architects/module-federation --project mf-shell
--port 4200 --type host

ng add @angular-architects/module-federation --project mf-login --port
    4201 --type remote

ng add @angular-architects/module-federation --project mf-table --port
    4202 --type remote

ng add @angular-architects/module-federation --project mf-dich-spesa
   --port 4203 --type remote

ng generate library commons-lib
```

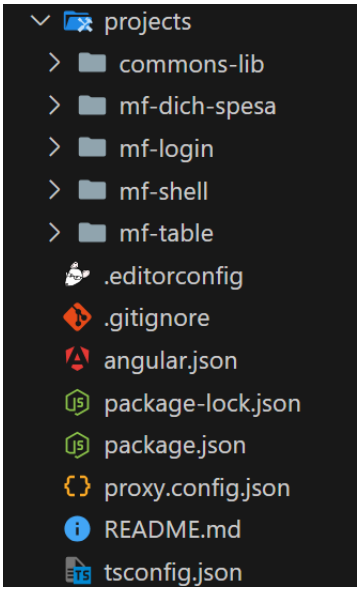

Figure 36: Microfrontend project composition

Here is a brief overview of the content of each project:

- **Component 1: mf-login**
  - **Description:** This project handles all user authentication procedure.
  - **Functionality:** Provides secure authentication through login form, handles authentication tokens, and manages user sessions. (UI as figure 27).
  - **Subcomponents/Services:** LoginComponent, LoginService.

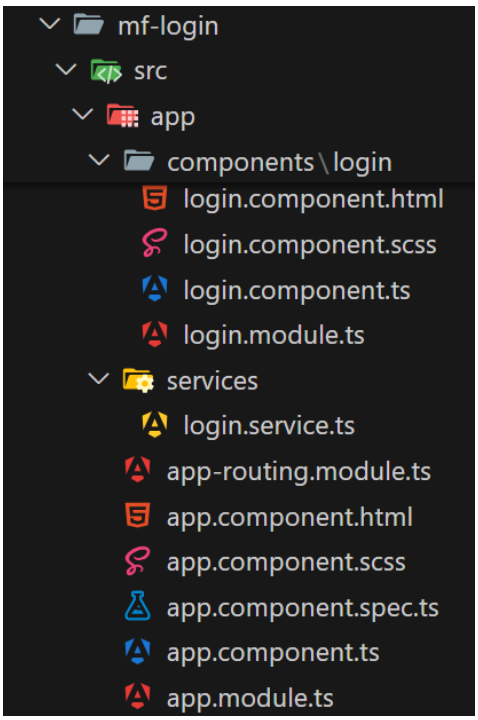

Figure 37: Microfrontend Login project

- **Component 2: mf-table**
  - **Description:** The table project manages all aspects of the table and its functionalities.
  - **Functionality:** The table dynamically updates whenever the user clicks on a level of the tree view. At a certain level, the user can download files. They can also delete files by clicking a button and confirming the action in a modal, or upload files through an upload modal. Upon successful completion of these actions, a success modal is displayed.(UI figure 29).
  - **Subcomponents/Services:** TableComponent, UploadComponent, ConfermaComponent, SucessComponent, UploadService, TableService.

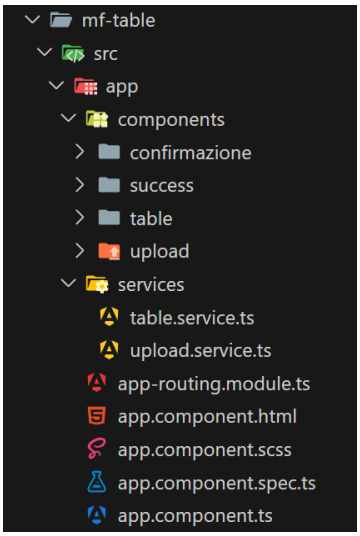

Figure 38: Microfrontend Table project

- **Component 3: mf-dich-spesa**
  - **Description:** This project manages an Excel-like spreadsheet within the application.
  - **Functionality:** It enables users to view an Excel spreadsheet and select different versions of the spreadsheet, each containing different data, by clicking a button.(UI figure 32).
  - **Subcomponents/Services:** DicSpesaComponent, DicSpesaService

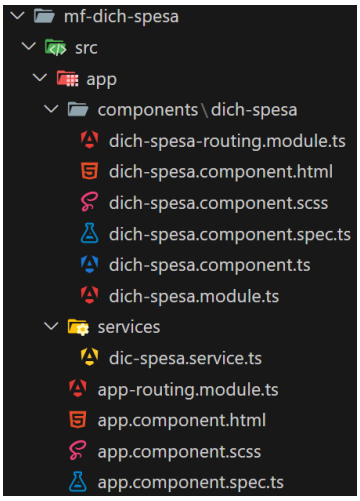

Figure 39: Microfrontend Dic Spesa project

- **Component 4: mf-shell**
  - **Description:** This project contains the navbar and the treeview, serving as the host for embedding all other projects.
  - **Functionality:** It manages the treeview by retrieving and displaying the necessary data. Additionally, it includes the router outlet and routes , figure 41, which dynamically replace the content with that of the other remote projects when the corresponding route is selected.
  - **Subcomponents/Services:** NavBarComponent, TreeViewComponent, TreeViewService.

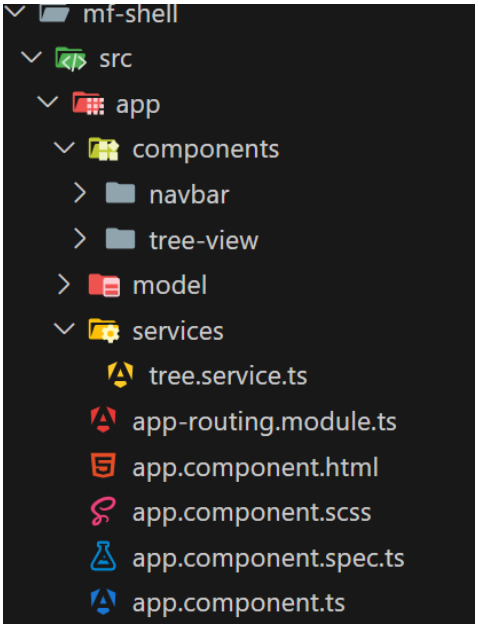

Figure 40: Microfrontend Shell project

```
const routes: Routes = [

  { path: '', pathMatch: 'full', redirectTo: 'login',
    data: { view: VIEW.LOGIN }
  },
  { path: 'login',
    loadChildren: () =>
      import('mfLogin/LoginModule').then((m) => m.LoginModule).catch(e
    => console.log(e)),
      data: { view: VIEW.LOGIN }
  },
  { path: 'table',
    loadChildren: () =>
      import('mfTable/TableModule').then((c) => c.TableModule).catch(e
    => console.log(e)),
      data: { view: VIEW.TABLE }
  },
  {  path: 'ds/:id',
    loadChildren: () =>
      import('mfDichSpesa/DichSpesaModule').then((c) => c.
   DichSpesaModule).catch(e => console.log(e)),
      data: { view: VIEW.DICH_SPESA }
  },
];
```

```
<app-navbar></app-navbar>

<ng-container>
    <div class="d-flex">
        <ng-container *ngIf="currentPath !== paths.login">
            <div class="col-2 position-relative d-inline-block">
                <app-tree-view></app-tree-view>
            </div>
        </ng-container>

        <div class="col-10 bg-light position-relative d-inline-block">
            <div class="container-fluid p-3 d-flex flex-column">
                <router-outlet></router-outlet>
            </div>
        </div>
    </div>
</ng-container>
```

Figure 41: Microfrontend App.component.html content

8.3.3 Communication Between Components commons-lib

The components communicate with each other using the commons-lib library shown in figure 42, which serves as a bridge to share data and trigger events across different microfrontends. This library facilitates seamless interaction and data exchange, ensuring that changes in one component can be reflected in another without tightly coupling them. The commons-lib plays a crucial role in the microfrontend architecture by providing a centralized repository for shared functionalities and services. Let us have a more detailed look on how this library works:

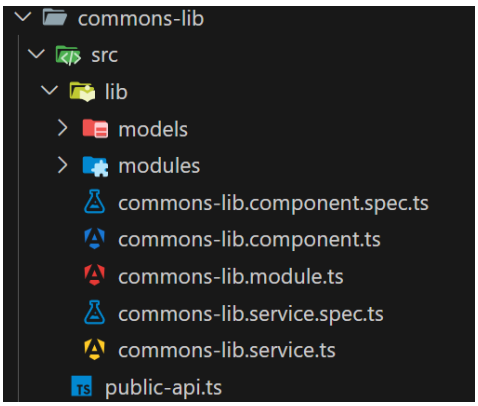

Figure 42: Content of the commons-lib library

Packages and data to share: We previously discussed how libraries can share functionalities like data packages, functions, services, and interfaces. Since our microfrontend project, `mf-shell`, relies on Observables within its service, we need the `rxjs` library. To make it available across projects, we'll install it using `npm` in the `commons-lib` (figure 43). This exposes `rxjs` for use in all our projects.

In addition, we need to declare a set of interfaces or data models that we are going to use in our microfrontend projects, so we will need to export them outside the library. These interfaces are: `Bando` and `UploadFileInfo`, shown in figure 44.

Lastly, we need to expose our data service, the main part that will be responsible for managing the flow of data between projects. To do this, we must create a service in the library, `CommonsLibService`, which will contain two objects (figure 45) of type `BehaviorSubject`: one of type `Boolean`, which serves to indicate that the view must be updated after an event (such as modify an element, upload a new one, or delete an existing one) and another of type `Object` that has two properties: `bando` (of type `Bando`) and another property `bandi` (formed by an array of objects of type `Bando`). This last

Figure 43: package.json of the commons-lib

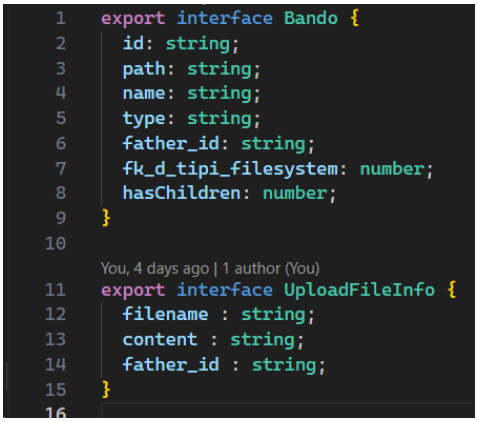

Figure 44: Interfaces of the commons-lib

`BehaviorSubject` dynamically stores the data that is loaded into our microFrontend `mf-shell`, in the `TreeViewComponent` component, and that can be consumed at the same time by the microfrontend `mf-table`, in charge of displaying through a table, the nodes that are selected. In this way, we are communicating and sharing information between two microfrontends, through a service implemented within the library.

Figure 45: Common Library Service

Expose data to the outside: In order for a library with its content to be used within other projects, it is necessary to export this data. When we create a library in Angular, a file called `public-api.ts` is created. This file contains a list of paths to the files we want to export:

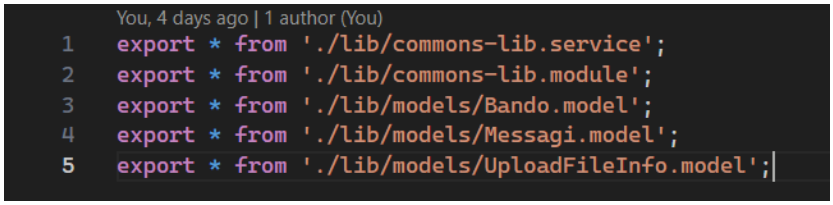

Figure 46: public-api.ts content

Now, the next step is to import the library into the projects where we need to implement it. At this point, we have two options:

- Add a momentary configuration in the `ts.config.json` file of the base project, to indicate that we must import the library as a project, that is, reference the library without having to build it and install it.
- Generate the build, package, and install the library in each project.

Since we are developing the library and need to make changes to it all the time, the best option is to add a temporary configuration in the `ts.config.json` file to indicate that it makes use of the library by directly importing the `public-api.ts` file of the library. Now, we have created a reference called

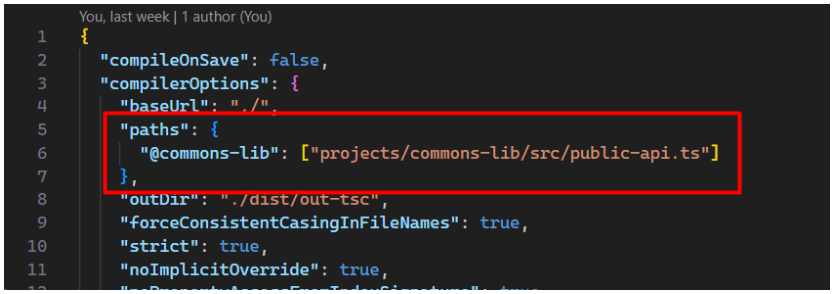

Figure 47: tconfig.json content

`@commons-lib`, which directly imports the library's public file, which exposes its modules, services, and interfaces.

Import the library into Microfrontend projects: Similar to any external library, we need to integrate it into the project where it's used. This can be done through installation or import. Since we're using Webpack, including the library requires an update to the Webpack configuration file. This configuration tells Webpack to handle the library during the bundling process. Without this step, the microFrontend wouldn't recognize the library. The following example demonstrates this configuration for the `mf-shell` microFrontend. We need to remember to repeat this process for any other microFrontend that utilizes the library.

Figure 48: Importing library in mf-shell project

Usage of the library in Microfrontend projects: As a last step, we have to import the functions we need for our project from the library. For example, in the `TreeViewComponent` component, we need to import the `Bando` interface and the `CommonsLibService` service from the library.

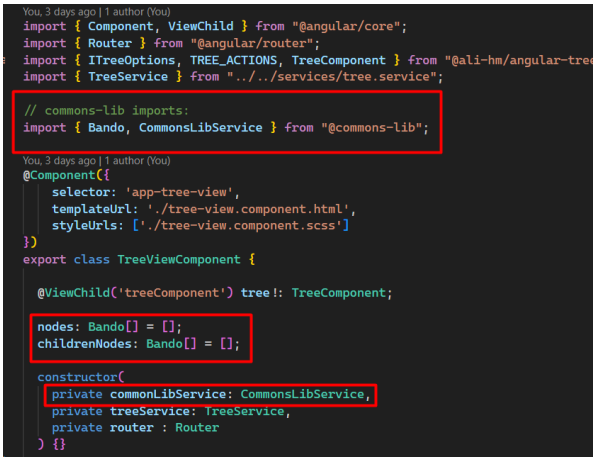

Figure 49: Library usage in tree view ui component

In this specific component, we can update the list of nodes through the `subjectNodes` property declared in the `CommonsLibService` service of the library and update the table view through the `updateView` property.

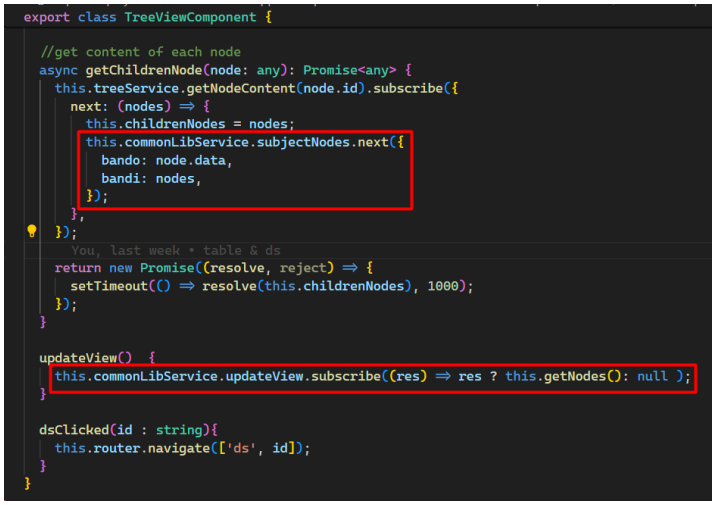

Figure 50: Library usage in tree view ui component

By leveraging commons-lib, the application achieves a high level of modularity and reusability, reducing duplication of code and ensuring consistent behavior across all microfrontends. This approach not only simplifies development and maintenance but also enhances the overall performance and scalability of the application.

8.4 Cypress test suites on Microfrontend projects

After implementing the application using the microfrontend architecture, end-to-end (e2e) testing was essential. Since each microfrontend component is in a separate project, the testing was conducted independently within each project.

8.4.1 Setup and Installation

The setup process involved installing Cypress in the main workspace and then in each individual project to allow for targeted testing. First, Cypress was installed in the main workspace using the following command:

```
npm i cypress
```

This command adds Cypress as a dependency in the workspace, enabling the execution of Cypress tests across different projects. Next, to install Cypress in each separate microfrontend project, the following command was executed:

```
npm i @cypress/schematic
```

This command integrates Cypress into each microfrontend project, ensuring that e2e tests can be run independently within each project. By setting up Cypress in this manner, we can effectively isolate and test the functionality of each microfrontend component while maintaining a consistent testing environment across the entire application.

8.4.2 Test Suites

Here is a detailed overview of some of the content of the tests made on two microfrontend projects, `mf-login` and `mf-table`:

- **Test Suite for `mf-login` Component:**
  One test suite `login.cy.ts` shown in figure 51, includes a series of tests to verify the functionality of the login page. Each test ensures that specific elements and behaviors are working correctly:

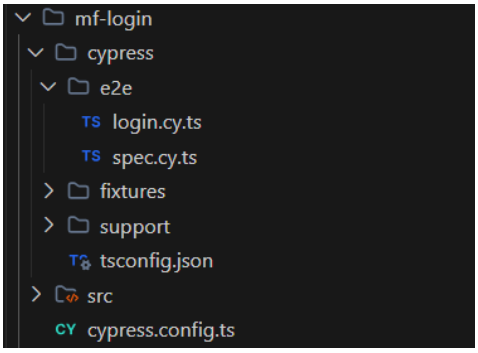

Figure 51: Test suite on mf-login

```
describe('Login Page', () => {
  // This block runs before each test to visit the login page
  beforeEach(() => {
    cy.visit('localhost:4200/login');
  });

  // 1) Test to check if the login form renders correctly
  it('should render the login form', () => {
    cy.get('form').should('be.visible'); // Verifies that the form is
   visible
    cy.get('input[name="username"]').should('be.visible'); // Verifies
    that the username input is visible
```

```
    cy.get('input[name="password"]').should('be.visible'); // Verifies that the password input is visible
  });

  // 2) Test to check successful login with correct credentials
  it('should log in successfully with correct credentials', () => {
    cy.intercept('POST', '/api/login', { statusCode: 200 }).as('loginRequest'); // Intercepts the login API call and mocks a successful response
    cy.get('input[name="username"]').type('validUser'); // Enters a valid username
    cy.get('input[name="password"]').type('validPassword'); // Enters a valid password
    cy.get('button[type="submit"]').click(); // Clicks the submit button
    cy.wait('@loginRequest').its('response.statusCode').should('eq', 200); // Waits for the API response and checks if the status code is 200
  });

  // 3) Test to check error message display on failed login attempt
  it('should show error message on failed login attempt', () => {
    cy.intercept('POST', '/api/login', { statusCode: 401 }).as('loginRequest'); // Intercepts the login API call and mocks a failed response
    cy.get('input[name="username"]').type('invalidUser'); // Enters an invalid username
    cy.get('input[name="password"]').type('invalidPassword'); // Enters an invalid password
    cy.get('button[type="submit"]').click(); // Clicks the submit button
    cy.wait('@loginRequest').its('response.statusCode').should('eq', 401); // Waits for the API response and checks if the status code is 401
    cy.get('.error-message').should('be.visible').and('contain', 'Invalid username or password'); // Checks if the error message is visible and contains the correct text
  });

  // 4) Test to check if the submit button is disabled while processing the login request
  it('should disable the submit button while processing', () => {
    cy.intercept('POST', '/api/login', { statusCode: 200 }).as('loginRequest'); // Intercepts the login API call and mocks a successful response
    cy.get('input[name="username"]').type('validUser'); // Enters a valid username
    cy.get('input[name="password"]').type('validPassword'); // Enters a valid password
    cy.get('button[type="submit"]').click(); // Clicks the submit button
    cy.get('button[type="submit"]').should('be.disabled'); // Checks if the submit button is disabled
    cy.wait('@loginRequest'); // Waits for the API response
    cy.get('button[type="submit"]').should('not.be.disabled'); // Checks if the submit button is enabled after the response
```

```
  });

  // 5) Test to check if validation messages are shown for empty
   fields
  it('should show validation messages for empty fields', () => {
    cy.get('button[type="submit"]').click(); // Clicks the submit
   button without filling in the fields
    cy.get('input[name="username"]:invalid').should('have.length', 1);
    // Checks if the username input is invalid
    cy.get('input[name="password"]:invalid').should('have.length', 1);
    // Checks if the password input is invalid
  });
});
```

Explanation of each test:

1. Rendering the Login Form: This test checks if the login form, along with the username and password input fields, is visible when the login page is loaded.
2. Successful Login with Correct Credentials: This test simulates a successful login by intercepting the login API request and returning a 200 status code. It ensures that the login process works correctly with valid credentials.
3. Error Message on Failed Login Attempt: This test simulates a failed login attempt by intercepting the login API request and returning a 401 status code. It verifies that an error message is displayed when invalid credentials are used.
4. Disabling Submit Button While Processing: This test ensures that the submit button is disabled while the login request is being processed, preventing multiple submissions. It then checks if the button is re-enabled after the after the request completes.
5. Validation Messages for Empty Fields: This test verifies that appropriate validation messages are shown when the user tries to submit the form with empty username and password fields.

- **Test Suite for `mf-table` Component:**
  Several test suites, shown in figure 53, have been conducted on this project because it has many functionalities. We will explore in details four of those test suites:

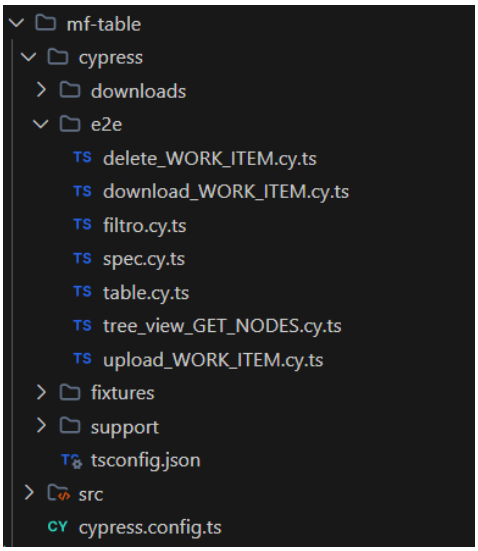

Figure 52: Test suite on mf-table

– **Test Suite** `download_work_item`**:**

```
describe('Tree View DOWNLOAD WORK ITEM', () => {
    beforeEach(() => {
        cy.visit('http://localhost:4200/table');
    });

    it('TreeViewComponents DOWNLOAD FIRST ITEM (Exist Control)', () =>
    {
        cy.get('tree-node').eq(0).click().as('getChildren1');
        cy.get('tree-node-children').eq(0).click().as('getChildren2').
   click();
        cy.get('tree-node-children tree-node').eq(0).click().as('
   getChildren3').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
   node-collection tree-node').eq(0).as('getChildren4').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
   node-collection tree-node tree-node-children tree-node-collection
   tree-node').eq(1).as('getChildren5').click();
        cy.get('table tbody tr').eq(6).find('td').eq(0).find('div').eq
   (1).find('span').eq(1).click().as('download[0]');
        cy.readFile(`C://Users/maria/Downloads/prova_3copy.txt`).
   should('exist');
    });
});
```

Explanation of the test: This test simulates the process of navigating through a tree view structure and downloading the first item. The test verifies that the item is successfully downloaded by checking for the existence of the downloaded file on the local file system.

– **Test Suite** `upload_work_item`**:**

```
describe('Tree View DELETE WORK ITEM', () => {
    beforeEach(() => {
        cy.visit('http://localhost:4200/table');
    });
    it('TreeViewComponents OPEN WORK FOLDER - UPLOAD ITEM', () => {
        const pathFile = 'C://Users/maria/Downloads/EXAMPLE.pdf';
        cy.get('tree-node').eq(0).click().as('getChildren1');
        cy.get('tree-node-children').eq(0).click().as('getChildren2').
   click();
        cy.get('tree-node-children tree-node').eq(0).click().as('
   getChildren3').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
   node-collection tree-node').eq(0).as('ds_1_FOLDER').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
   node-collection tree-node tree-node-children tree-node-collection
   tree-node').eq(1).as('WORK_FOLDER').click();
        cy.get('button[id=upload]').click().as('UPLOAD_BUTTON_CLICK');
        cy.get('input[type=file]').selectFile(pathFile);
        cy.get('wise-upload button[id=sendUpload]').click().as('
   CARICA_FILE');
        cy.readFile(pathFile).should('exist');
    });});
```

Explanation of the test: This test simulates the process of opening a specific work folder in the tree view and uploading a file to it. It verifies that the file upload process is successful by checking for the existence of the uploaded file on the local file system.

– **Test Suite** `tree_view_GET_NODES`**:**

```
describe('Tree View Tests', () => {
    beforeEach(() => {
        cy.visit('http://localhost:4200/table');
    });

    it('TreeViewComponents GET FIRST LEVEL CHILDREN', () => {
        cy.get('tree-node').eq(0).click().as('getChildren1');
        cy.get('tree-node-children').eq(0).click().as('getChildren2').
    click();
        cy.get('tree-node-children tree-node').eq(0).click().as('
    getChildren3').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
    node-collection tree-node').eq(0).as('getChildren4').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
    node-collection tree-node tree-node-children tree-node-collection
    tree-node').eq(1).as('getChildren5').click();

        cy.get('table tbody tr').eq(0).find('td').eq(0).find('div').eq
    (1).find('span').eq(0).click().as('delete[0]');
    });

    it('TreeViewComponents GET SECOND LEVEL CHILDREN', () => {
        cy.get('tree-node').eq(0).click().as('getChildren1');
        cy.get('tree-node-children').eq(0).click().as('getChildren2').
    click();
    });

    it('TreeViewComponents GET WORK LEVEL', () => {
        cy.get('tree-node').eq(0).click().as('getChildren1');
        cy.get('tree-node-children').eq(0).click().as('getChildren2').
    click();
        cy.get('tree-node-children tree-node').eq(0).click().as('
    getChildren3').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
    node-collection tree-node').eq(0).as('getChildren4').click();
        cy.get('tree-node-children tree-node tree-node-children tree-
    node-collection tree-node tree-node-children tree-node-collection
    tree-node').eq(1).as('getChildren5').click();
    });
});
```

Explanation of each test:

1. **Tree View Tests:** This test suite verifies different aspects of the tree view functionality within the table page.
2. **GET FIRST LEVEL CHILDREN:** This test navigates through the tree view to get to the first level children and performs a click action. It also performs a delete action on the first item in the table.
3. **GET SECOND LEVEL CHILDREN:** This test navigates through the tree view to get to the second level children.

4. **GET WORK LEVEL:** This test navigates through the tree view to get to the work level and verifies the navigation.

– **Test Suite `filtro`:**
This test suite includes a series of tests to verify the functionality of the filter form in the table page. Each test ensures that specific elements and behaviors are working correctly:

```
describe('Fitro Form Test', () => {
    beforeEach(() => {
        cy.visit('localhost:4200/table');
    });

    // 1)
    it('should fill out the form and submit', () => {
        cy.get('input[name="numDichSpesa"]').type('12345');
        cy.get('input[name="numDomanda"]').type('67890');
        cy.get('input[name="numBenef"]').type('ABCDE');
        cy.get('button[type="submit"]').click();
    });
    // 2)
    it('should reset the form fields when clicking "Azzera filtri"', () => {
        cy.get('input[name="numDichSpesa"]').type('12345');
        cy.get('input[name="numDomanda"]').type('67890');
        cy.get('input[name="numBenef"]').type('ABCDE');
        cy.contains('Azzera filtri').click();
        cy.get('input[name="numDichSpesa"]').should('have.value', '');
        cy.get('input[name="numDomanda"]').should('have.value', '');
        cy.get('input[name="numBenef"]').should('have.value', '');
    });
    // 3)
    it('should clear all input fields when Azzera filtri button is clicked', () => {
        cy.get('#dsInput').type('12345');
        cy.get('#dsNumDomandaInput').type('54321');
        cy.get('#dsBenefInput').type('Benef123');
        cy.get('button').contains('Azzera filtri').click();
        cy.get('#dsInput').should('have.value', '');
        cy.get('#dsNumDomandaInput').should('have.value', '');
        cy.get('#dsBenefInput').should('have.value', '');
    });
    // 4)
    it('should allow text input in NumDichSpesa field', () => {
        cy.get('#dsInput').type('12345').should('have.value', '12345');
    });
    // 5)  it('should allow text input in NumDomanda field', () => {
        cy.get('#dsNumDomandaInput').type('54321').should('have.value', '54321');
    });
    // 6)
    it('should allow text input in NumBenef field', () => {
        cy.get('#dsBenefInput').type('Benef123').should('have.value', 'Benef123');
    });
});
```

Explanation of each test:

1. **Fitro Form Test:** This test suite verifies different aspects of the filter form functionality within the dichiarazione di spesa page.
2. **Should fill out the form and submit:** This test fills out the form fields for numDichSpesa, numDomanda, and numBenef and submits the form.
3. **Should reset the form fields:** This test fills out the form fields and then clicks "Azzera filtri" to reset the form fields to their initial state.
4. **Should clear all input fields:** This test fills out the form fields and clicks "Azzera filtri" to clear all input fields.
5. **Should allow text input in NumDichSpesa field:** This test verifies that text input is allowed in the NumDichSpesa field.
6. **Should allow text input in NumDomanda field:** This test verifies that text input is allowed in the NumDomanda field.
7. **Should allow text input in NumBenef field:** This test verifies that text input is allowed in the NumBenef field.

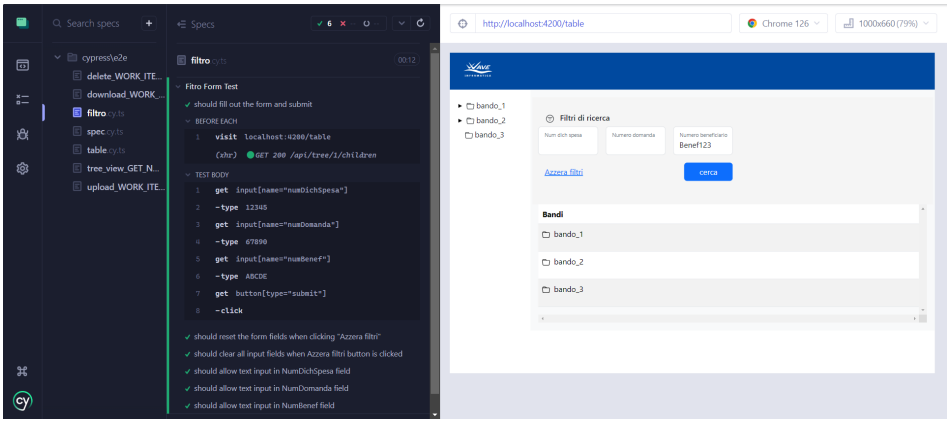

Figure 53: Snap-shot of filtro.cy test suite

9 Conclusion

The journey of software architecture has evolved significantly, transitioning from monolithic structures to highly modular and scalable microservices and microfrontends. This thesis explored the critical aspects and motivations behind this evolution, presenting a detailed analysis of the advantages and challenges associated with each architectural style. The transition from monolithic to microservices architecture marks a significant shift in software development. Monolithic applications, while straightforward in their initial development phases, often lead to complexities in scaling, maintenance, and deployment. The thesis highlighted the foundational features of microservices, emphasizing their ability to enhance scalability, improve fault isolation, and enable continuous deployment. The migration strategies discussed provide a comprehensive roadmap for organizations considering this transition.

The concept of microfrontends has emerged as a solution to the limitations of traditional monolithic front-end development. By enabling independent development, deployment, and scaling of individual front-end components, microfrontends offer a modular approach that aligns with the principles of microservices. This thesis detailed the various approaches to building graphical user interfaces using microfrontends, their relationship with microservices architectures, and the unique challenges they present, such as managing inter-module dependencies and ensuring consistent user experiences. The advantages, including improved development velocity and better team autonomy, underscore the growing adoption of this architecture in modern web applications.

Webpack has become a cornerstone tool in modern web development, facilitating efficient module bundling and dependency management. The thesis covered the primary advantages of using Webpack, such as its capability to optimize asset delivery and streamline the build process. Additionally, the mechanics of Webpack were dissected to provide a clear understanding of its configuration and operation. The practical application of Webpack in a microfrontend setup, particularly with Webpack 5's Module Federation, was illustrated through a detailed walkthrough, demonstrating its role in achieving dynamic, runtime integration of microfrontend modules.

Robust software testing is crucial for maintaining the quality and reliability of applications. The thesis explored various software testing methodologies, including static and dynamic techniques, and strategies for effective testing. The limitations of manual testing were contrasted with the benefits of automated testing frameworks, highlighting the efficiency and coverage automated tests provide. Cypress, a modern end-to-end testing framework, was discussed in depth. Its declarative syntax, automatic waiting capabilities, and integration with CI/CD pipelines make it a powerful tool for developers.

The practical application of the discussed concepts was exemplified through a detailed company use case scenario. This section demonstrated s explored the re-implementation of an existing single-page application (SPA) using the microfrontend architecture, and detailed testing of each microfrontend component. The primary objective was to leverage the benefits of microfrontend architecture, such as improved modularity, scalability, and maintainability, to enhance the overall performance and flexibility of the application.

Through this work, we demonstrated the effectiveness of the microfrontend architecture in modernizing and optimizing web applications. The transformation of the SPA into a microfrontend-based application resulted in a more modular and maintainable codebase, which can be developed and deployed independently. This approach also facilitated better team collaboration, as different teams could work on separate microfrontends without causing conflicts.

In conclusion, the microfrontend architecture offers significant advantages for large-scale web applications by promoting modularity, scalability, and maintainability. The successful re-implementation of the SPA into a microfrontend application in this thesis serves as a testament to these benefits. Future work could focus on further optimizing the integration process, exploring alternative technologies, and conducting performance benchmarks to quantify the improvements achieved through this architectural shift.

A Cypress Example Code and Detailed Comments

```
/// <reference types="cypress" />


describe('example to-do app', () => {
  beforeEach(() => {
    // Cypress starts out with a blank slate for each test
    // so we must tell it to visit our website with the `cy.visit()` command.
    // Since we want to visit the same URL at the start of all our tests,
    // we include it in our beforeEach function so that it runs before each test
    cy.visit('https://example.cypress.io/todo')
  })

  it('displays two todo items by default', () => {
    // We use the `cy.get()` command to get all elements that match the selector.
    // Then, we use `should` to assert that there are two matched items,
    // which are the two default items.
    cy.get('.todo-list li').should('have.length', 2)

    // We can go even further and check that the default todos each contain
    // the correct text. We use the `first` and `last` functions
    // to get just the first and last matched elements individually,
    // and then perform an assertion with `should`.
    cy.get('.todo-list li').first().should('have.text', 'Pay electric bill')
    cy.get('.todo-list li').last().should('have.text', 'Walk the dog')
  })

  it('can add new todo items', () => {
    // We'll store our item text in a variable so we can reuse it
    const newItem = 'Feed the cat'

    // Let's get the input element and use the `type` command to
    // input our new list item. After typing the content of our item,
    // we need to type the enter key as well in order to submit the input.
    // This input has a data-test attribute so we'll use that to select the
    // element in accordance with best practices:
    // https://on.cypress.io/selecting-elements
    cy.get('[data-test=new-todo]').type(`${newItem}{enter}`)

    // Now that we've typed our new item, let's check that it actually was added to the list.
    // Since it's the newest item, it should exist as the last element in the list.
    // In addition, with the two default items, we should have a total of 3 elements in the list.
    // Since assertions yield the element that was asserted on,
```

```
43      // we can chain both of these assertions together into a single
        statement.
44      cy.get('.todo-list li')
45        .should('have.length', 3)
46        .last()
47        .should('have.text', newItem)
48    })
49
50    it('can check off an item as completed', () => {
51      // In addition to using the `get` command to get an element by
        selector,
52      // we can also use the `contains` command to get an element by its
        contents.
53      // However, this will yield the <label>, which is lowest-level
        element that contains the text.
54      // In order to check the item, we'll find the <input> element for
        this <label>
55      // by traversing up the dom to the parent element. From there, we
        can `find`
56      // the child checkbox <input> element and use the `check` command
        to check it.
57      cy.contains('Pay electric bill')
58        .parent()
59        .find('input[type=checkbox]')
60        .check()
61
62      // Now that we've checked the button, we can go ahead and make
        sure
63      // that the list element is now marked as completed.
64      // Again we'll use `contains` to find the <label> element and then
        use the `parents` command
65      // to traverse multiple levels up the dom until we find the
        corresponding <li> element.
66      // Once we get that element, we can assert that it has the
        completed class.
67      cy.contains('Pay electric bill')
68        .parents('li')
69        .should('have.class', 'completed')
70    })
71
72    context('with a checked task', () => {
73      beforeEach(() => {
74        // We'll take the command we used above to check off an element
75        // Since we want to perform multiple tests that start with
        checking
76        // one element, we put it in the beforeEach hook
77        // so that it runs at the start of every test.
78        cy.contains('Pay electric bill')
79          .parent()
80          .find('input[type=checkbox]')
81          .check()
82      })
83
84      it('can filter for uncompleted tasks', () => {
85        // We'll click on the "active" button in order to
86        // display only incomplete items
```

```
87        cy.contains('Active').click()
88
89        // After filtering, we can assert that there is only the one
90        // incomplete item in the list.
91        cy.get('.todo-list li')
92          .should('have.length', 1)
93          .first()
94          .should('have.text', 'Walk the dog')
95
96        // For good measure, let's also assert that the task we checked
      off
97        // does not exist on the page.
98        cy.contains('Pay electric bill').should('not.exist')
99      })
100
101     it('can filter for completed tasks', () => {
102       // We can perform similar steps as the test above to ensure
103       // that only completed tasks are shown
104       cy.contains('Completed').click()
105
106       cy.get('.todo-list li')
107         .should('have.length', 1)
108         .first()
109         .should('have.text', 'Pay electric bill')
110
111       cy.contains('Walk the dog').should('not.exist')
112     })
113
114     it('can delete all completed tasks', () => {
115       // First, let's click the "Clear completed" button
116       // `contains` is actually serving two purposes here.
117       // First, it's ensuring that the button exists within the dom.
118       // This button only appears when at least one task is checked
119       // so this command is implicitly verifying that it does exist.
120       // Second, it selects the button so we can click it.
121       cy.contains('Clear completed').click()
122
123       // Then we can make sure that there is only one element
124       // in the list and our element does not exist
125       cy.get('.todo-list li')
126         .should('have.length', 1)
127         .should('not.have.text', 'Pay electric bill')
128
129       // Finally, make sure that the clear button no longer exists.
130       cy.contains('Clear completed').should('not.exist')
131     })
132   })
133 })
```

References

[1] Nahid Anwar and Susmita Kar. "Review Paper on Various Software Testing Techniques & Strategies". Version 1.0. In: *Global Journal of Computer Science and Technology : C Software & Data Engineering* 19.2 (2019).

[2] Kevin Ball. *Microfrontends: The Good, the Bad, and the Ugly*. Accessed: 2020-05-17. 2019. URL: https://zendev.com/2019/06/17/microfrontends-good-bad-ugly.html.

[3] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1995.

[4] Boris Beizer. *Software Testing Techniques*. 2nd. New York, NY, USA: Van Nostrand Reinhold Computer, 1990.

[5] Bitsrc.io. *You Don't Need Another Library to Compose Micro Frontends at Run Time*. Accessed: 2020-05-17. 2020. URL: https://blog.bitsrc.io/you-dont-need-another-library-to-compose-micro-frontends-at-run-time-e803077ade67.

[6] Mohamed Bouzid. *Webpack for beginners*. https://leanpub.com/webpack-for-beginners. 2019.

[7] Kevin Brown. "Apply the Strangler Application pattern to microservices applications". In: (2017). URL: https://developer.ibm.com/technologies/microservices/articles/cl-strangler-application-pattern-microservices-apps-trs/.

[8] BugBug. *Cypress Best Practices - Basics*. Year of Publication. URL: %5Curl%7Bhttps://bugbug.io/blog/testing-frameworks/cypress-best-practices/#cypress-best-practices---basics%7D.

[9] C. Carneiro Jr and T. Schmelmer. In: *Microservices From Day One*. Springer, 2016, pp. 177–184.

[10] Chris Coyier. *Micro Frontends*. Accessed: 2020-05-17. 2019. URL: https://css-tricks.com/micro-frontends/.

[11] E. Dijkstra. *Notes On Structured Programming*. Tech. rep. 70-WSK-03. The Netherlands: Dept. of Mathematics, Technological University of Eindhoven, Apr. 1970.

[12] Cypress Documentation. *Continuous Integration: Introduction*. https://docs.cypress.io/guides/continuous-integration/introduction.

[13] Cypress Documentation. *What is Cypress?* https://docs.cypress.io/guides/overview/why-cypress. 2024.

[14] Cypress Documentation. *Writing and Organizing Tests*. https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests.

[15] Martin Fowler. *Monolith First*. 2015. URL: https://martinfowler.com/bliki/MonolithFirst.html.

[16] Jonas FRITZSCH et al. "Microservices Migration in Industry: Intentions, Strategies, and Challenges". In: *2019*. DOI: 10.1109/ICSME.2019.00081.

[17] Michael Geers. *Micro Frontends: Extending the Microservice Idea to Frontend Development*. Accessed: 2020-05-17. 2019. URL: https://micro-frontends.org/.

[18] Phodal Huang. *Micro-frontend Architecture in Action with Six Ways*. Accessed: 2020-05-17. 2019. URL: https://dev.to/phodal/micro-frontend-architecture-in-action-4n60.

[19] Cam Jackson. *Micro Frontends*. Accessed: 2020-05-17. 2019. URL: https://martinfowler.com/articles/micro-frontends.html.

[20] Benjamin Johnson. *Exploring Micro-Frontends*. Accessed: 2020-05-17. 2018. URL: https://medium.com/@benjamin.d.johnson/exploring-micro-frontends-87a120b3f71c.

[21] Jovanovic and Irena. "Software Testing Methods and Techniques". In: *Technical Report* (May 2008).

[22] Miklós Kain and Péter Mihók. *Transforming monolithic architecture towards microservice architecture*. CORE. 2019. URL: https://core.ac.uk/reader/157587910.

[23] Sandeep Kakarh. *Advantages and Disadvantages of Cypress End-to-End Testing Tool Before Choosing It as Your Testing Framework*. https://skakarh.medium.com/advantages-and-disadvantages-of-cypress-end-to-end-testing-tool-before-choosing-it-as-your-347b6436dec8.

[24] Mohd. Ehmer Khan. "Different Forms of Software Testing Techniques for Finding Errors". In: *IJCSI International Journal of Computer Science Issues* 7.3 (1 May 2010).

[25] A. Kharenko. *Monolithic vs Microservices Architecture*. 2015. URL: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59.

[26] Komodor. *Monolith to Microservices: 5 Strategies, Challenges, and Solutions*. https://komodor.com/learn/monolith-to-microservices-5-strategies-challenges-and-solutions/.

[27] Amit Kothari. *What is Micro Frontend?* Accessed: 2020-05-17. 2017. URL: https://hub.packtpub.com/what-micro-frontend/.

[28] J. Lewis and M. Fowler. *Microservices: A Definition of This New Term*. 2014. URL: https://martinfowler.com/articles/microservices.html.

[29] Chia-Yu LI, Shang-Pin MA, and Tsung-Wen LU. "Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System". In: *2020 International Computer Symposium (ICS)*. 2020, pp. 519–524. DOI: 10.1109/ICS51289.2020.00107.

[30] Detlef Mertins, Rodolphe El-Khoury, and Rodolfo Machado. *Monolithic Architecture (Architecture & Design)*. Prestel Pub, 1995.

[31] *Micro-frontends: Application of Microservices to Web Front-ends*. See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/351282486. 2021. URL: https://www.researchgate.net/publication/351282486.

[32] Microsoft. *Domain Driven design*. https://learn.microsoft.com/en-us/azure/architecture/microservices/migrate-monolith.

[33] S. Mikowski and J. C. Powell. *Single Page Web Applications*. Shelter Island: Manning, Sept. 2013.

[34] Alejandro Morales. *Cypress: The Future of Web Testing*. https://www.theseus.fi/bitstream/handle/10024/806779/Morales_Alejandro.pdf?sequence=2. 2023.

[35] Glenford J. Myers. *The Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979.

[36] Author Name. "The Impact of Manual and Automatic Testing on Software Testing Efficiency and Effectiveness". In: *Indian Journal od science and research* (2023), p. 10.

[37] Sam Newman. *Building Microservices*. United States of America: O'Reilly Media, Inc., 2015.

[38] Ron Patton. *Software Testing*. Sams Publishing, 2006.

[39] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014. Chap. 7.

[40] C. Richardson. *Microservices*. 2015. URL: http://microservices.io.

[41] C. Richardson. *Monolithic Architecture*. 2015. URL: http://microservices.io/patterns/monolithic.html.

[42] Abhijit A. Sawant, Pranit H. Bari, and P. M. Chawan. "Software Testing Techniques and Strategies". In: *Department of Computer Technology, VJTI, University of Mumbai, INDIA* (2019).

[43] Sha. "Title of the Article". In: *Int. Journal of Engineering Research and Applications* 4.4 (Apr. 2014). Version 9, pp. 99–102. ISSN: 2248-9622. URL: http://www.ijera.com.

[44] Sharvishi9118. *How to Compose Micro Frontends at Build Time*. Accessed: 2020-05-17. 2020. URL: https://sharvishi9118.medium.com/how-to-compose-micro-frontends-at-build-time-c5e484a40e10.

[45] Bc. Jiří Široký. *From Monolith to Microservices: Refactoring Patterns*. https://is.muni.cz/th/f8zv4/Master_Thesis.pdf. 2021.

[46] International Journal of Soft Computing and Engineering (IJSCE). "A Research Study on Importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Model". In: *International Journal of Soft Computing and Engineering* 2.3 (July 2012). Retrieval Number: C0761062312 /2012©BEIESP. ISSN: 2231-2307.

[47] Richard Torkar. "Towards Automated Software Testing Techniques, Classifications and Frameworks". Doctoral Dissertation. Blekinge Institute of Technology, 2006. ISBN: 91-7295-089-7.

[48] Alan M. Turing. "Checking a Large Routine". In: *Report of a Conference on High-Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, UK. 1949.

[49] Udemy. *Cypress - Web Automation Testing from Zero to Hero*. https://www.udemy.com/course/cypress-web-automation-testing-from-zero-to-hero/learn/lecture/18179436.

[50] Velv. *Server Side Rendering*. Accessed: 2021-06-05. 2021. URL: https://www.velv.pt/updates/server-side-rendering.

[51] Juho Vepsalainen. *SurviveJS - Webpack: From Apprentice to Master*. English. 2nd ed. Paperback. CreateSpace Independent Publishing Platform, Feb. 2016, p. 284. URL: https://riptutorial.com/Download/webpack.pdf.

[52] Hulya VURAL and Murat KOYUNCU. "Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?" In: *IEEE Access* PP (2021), pp. 1–1. DOI: 10.1109/ACCESS.2021.3060895.

[53] *Webpack Concepts*. https://webpack.js.org/concepts/.

[54] Webpack Contributors. *Webpack - Module Federation*. https://webpack.js.org/concepts/module-federation/. accessed 2024.

[55] *Webpack Tutorial*. https:/riptutorial.com/Download/webpack.pdf.

[56] *webpack-5-ninja*. Udemy.

[57] C. Yang, C. Liu, and Z. Su. "Research and Application of Micro Frontends". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 490. 1. Apr. 2019, pp. 1–8.