Tampere University

Aapo Lempiö

ENABLING CHANGEABILITY WITH TYPESCRIPT AND MICROSERVICES ARCHITECTURE IN WEB APPLICATIONS

Faculty of Information Technology and Communication Sciences
M. Sc. Thesis
April 2023

ABSTRACT

Aapo Lempiö: Enabling changeability with typescript and microservices architecture in web applications
M.Sc. Thesis
Tampere University
Master's Degree Programme in Computer Science
April 2023

Changeability is a non-functional property of software that affects the length of its lifecycle. In this work, the microservices architectural pattern and TypeScript are studied through a literature review, focusing on how they enable the changeability of a web application.

The modularity of software is a key factor in enabling changeability. The microservices architectural pattern and the programming language TypeScript can impact the changeability of web applications with modularity. Microservices architecture is a well-suited architectural pattern for web applications, as it allows for the creation of modular service components that can be modified and added to the system individually. TypeScript is a programming language that adds a type system and class-based object-oriented programming to JavaScript offering an array of features that enable modularity.

Through discussion on relationships between the changeability of web applications and their three key characteristics, scalability, robustness, and security, this work demonstrates the importance of designing for change to ensure that web applications remain maintainable, extensible, restructurable, and portable over time. Combined, the microservices architecture and TypeScript can enhance the modularity and thus changeability of web applications.

Keywords: changeability, TypeScript, microservices, modularity

Contents

1 Introduction

The ability for software to be adaptable has grown to be a great advantage in recent decades, especially when web application development is considered. As change is a driving force in the market for web applications or even most software, it needs to be accommodated from the start, beginning at the architectural level. The extent to and the efficiency with which changes can be made to the software is called changeability [1].

Many factors affect software change, starting from choosing a pattern or patterns as the basis for the architectural design. It is an important decision to make early on. There are various architectural patterns each having its strengths and weaknesses. Microservices architecture is one such pattern. It has gained popularity in the recent decade as it has many suitable qualities from the standpoint of web applications. The programming languages that the software is programmed with also influences the changeableness of the software.

In this work, modularity is discussed as a preventative measure to software architecture erosion in web applications through a non-functional property of software called changeability. Web applications have certain characteristics that need to be accounted for when designing for changeability. The effects will be examined from the point of view of the microservices architectural pattern and TypeScript programming language.

TypeScript was chosen to reflect the impact that a programming language may have on the changeability of a web application as it is a commonly used language in the implementation of modern web applications. Also commonly used in the implementation of modern web applications is the microservices architectural pattern that will be examined in this work.

1.1 Research objectives and research questions

The **research questions** this thesis set out to answer are the following:

1. **RQ1**: “What enables the changeability of modern web applications?”
2. **RQ2**: "What changeabilities do TypeScript-based microservices architecture web applications have?”

To answer RQ1, web applications are defined as their own category of software. The changeability aspects that manifest in modern web applications will be identified through a literature review and analysis based on current research. As changeability is a key aspect that has a major influence on the whole products lifecycle, taking all its components into account beginning from the initialization until the end-of-life of the project will help to ensure that not only will work at launch but can keep up the pace as time goes on as requirements shift and environments evolve. There are other important factors to software longevity but as those do not fall in the scope of this thesis they will not be specifically inspected or identified in this work. To answer RQ1, the enablers of changeability in web applications architecture have to be identified and their relation to the changeability aspects has to be identified.

To answer RQ2, web application architecture is discussed from the perspective of change. Factors that enable change in a web application architecture are discussed from the point of view of using TypeScript and the microservices architectural pattern.

This thesis intends to summarize how TypeScript enables change when using the microservices architectural pattern and the impact of using the pattern over the changeability of web applications. First, the theory on these subjects will be studied through a literature review. The requirements for changeability in a web application and the architectural enablers for software change will be identified. The microservices architectural pattern will be examined and discussed in the context of web applications. TypeScript will be examined concerning changes in the structure of software. The aspects of changeability will be identified and the requirements that they impose on software will be discussed. After the theory has been presented, an analysis of the findings will be conducted. The purpose is to offer insight into what degree TypeScript as a programming

language makes change possible when implementing microservices and what solutions microservices architectural patterns enable in a web application.

The main purpose of this thesis is to give a broad general overview of changeability, web applications architectural considerations as well as considerations on the influence the choice of programming language has on the architecture from the changeability point of view. It is not intended as an exhaustive study on the subject areas as there are more things to consider in the scope of the topic of this thesis than could be meaningfully described in the context of this work due to the complexity of these issues.

1.2 Chapter outline

Below is the outline of the contents of this thesis and a brief description of what is to be discussed in each chapter, beginning from the next chapter:

- In Chapter 2, **Characteristics of modern web applications**, web applications are defined as their own type of software, having their unique set of characteristics.
- In Chapter 3, **Modularity and change,** the change-enabling techniques through modularity are introduced.
- In Chapter 4, **Microservices architectural pattern**, the microservices architectural pattern is introduced and discussed in the web application context, and how it enables change is revealed.
- Chapter 5, **TypeScript,** is an introduction to TypeScript and its relevant characteristics.
- In Chapter 6, **Changeability**, changeability is introduced as a non-functional property of software. Its four aspects are introduced and discussed.
- In Chapter 7, **Enabling changeability in TypeScript web applications with microservices architecture**, changeability in modern web applications is discussed by evaluating the relationship between TypeScript and the change enablers of software architecture in a microservices architecture web application.
- In Chapter 8, **Discussion,** this study and analysis will be introspectively discussed.

- In Chapter 9, **Conclusion**, all the previous chapters are summarized, and conclusions are drawn.

2 Characteristics of modern web applications

Web applications are a type of software that share similar features. A combination of characteristics commonly found in web applications has been identified. They usually require a multidisciplinary team to develop as there is a range of expertise required to produce it. [2] To name a few disciplines: systems analysis and design, software engineering, hypermedia and hypertext engineering, requirements engineering, human-computer interaction, user interface development, information engineering, information indexing and retrieval, testing, modeling and simulation, project management, graphic design and presentation [3]. In other words, developing web applications may require a sizeable development team.

Web applications are developed with technologies that are continuously being developed themselves. This means that usually the technologies used to produce them are frequently maintained. However, this also somewhat inconveniently means that an application might have to be updated as the technologies themselves are updated. This predisposes web applications to faults that stem from the changing environment, such as security vulnerabilities as new risks emerge, or previously neglected ones realize. This cycle has been described in the eight law in the Lehman's laws of software evolution [4]. The Lehman's laws will be discussed in Chapter 6.

As with any other software, enabling change is important in web applications. As hardware improves and web technologies advance, there will always be something to be improved upon. New requirements will surface one way or the other. If the impact that changes have on the growing complexity of a web application is not considered, it will have consequences that will potentially render the application unusable or unmaintainable. This phenomenon is called software erosion [5]. Software erosion has many aspects, one of them being the erosion of architecture. It's the cumulative damage that results from changes that have been made without fully considering the architectural effects [5]. It is an unavoidable consequence of continuous software development to some

extent [6], but it does not mean that it could not be stopped and even reversed. If not considered, the software will erode to the point at which there is no reason in trying to reverse the process as the cost of repair has risen too much. [5] Support for change should be built into the software and gradually maintained as a preventative measure instead of a reactive one.

2.1 Modern web applications

What constitutes software as a web application? For software to be a web application it must run on a browser be it desktop or mobile executed over a web platform [7]. A web application is not just a *website* or a *webpage* that comprises static text and styling and links between pages. Modern web applications are more complex than their not-so-distant ancestors as the web development scene has been evolving at a very high pace. They are distributed systems with dynamic information and service delivery [7]. A web page is described through HTML that the browser parses into the presented form. A web application uses a network of components and implements client-side programs and information handling, while HTML is also produced client-side. But what are the characteristics of the *modern* web applications of today and why?

In the context of this thesis, the term “modern” means something along the lines of “during the last ten years or so”. A variety of factors such as the emergence of smartphones in the late 2000s and the general advance in web technology led to the emphasis on web applications over native applications that were specifically designed for a single platform [8].

One of the drivers behind the design of web applications is getting as many users as possible to use the services provided by the application. In the 1990s and early 2000s when the web scene started its explosive growth journey, it was more than enough to have a website on the internet because the user interfaces were not as developed, and the users were used to the basic nature of software and hardware of the time. As time went on, and web technologies were continuously developed as was the hardware, more and more sophisticated websites and web applications became more commonplace. [9]

Coming to the 2010s and further on until the present, this led to the user experience becoming one of the most important factors in the design of an application besides

information security. Simply because there are abundant alternatives to almost any service on the internet. If a business wants its business to grow through the web application they are deploying, it must grow its user traffic. This can be aided by making the user experience appealing to the end user. [8]

Modern web applications differ from traditional native applications which can only be used on a single platform, meaning desktop or browser for example. Most web applications of today are designed in a way that the most used browsers and platforms are supported. It has evolved to be this way because if a platform or browser does not support a certain application, this can be viewed as a negative by the end user leading them elsewhere in search of service. Previously native applications gave much more room for innovation but in a limited context. Today's web applications are as capable in almost every regard as native applications. Web applications that have these traits are called *progressive web applications* and most new web applications are developed this way. A progressive web application will follow common web standards. Standardization is a powerful tool for developers that makes cross-platform support of the web application more trivial. [8] Web applications of today make use of a variety of packages and frameworks used locally and remotely that has been developed by a third party. This is called *component-based development,* and it lowers the amount of coding needed for a web application [10].

2.2 Architectural requirements of web applications for adaptability

Common quality attributes of a web application architecture include scalability, robustness, and security [2]. Scalability means the efficient use of infrastructure behind the web application in a way that makes it possible to handle user traffic on the web application even during times of peak use with a scalable back-end server, API, and database use. On the other hand, the application should be able to scale down also when it has negative peak traffic compared to the average amount of users. The number of users can vary greatly even within very short time ranges. This should be noted when designing a web application and analyzed later during production as not every web application should be able to handle as many simultaneous users as Amazon for example. [11] Cloud

computing has made it possible for almost every web application to be reasonably scalable because companies such as Amazon (AWS) and Microsoft (Azure) offer services for component-based development such as infrastructure as a service (IaaS), serverless computing (FaaS) and platform as a service (PaaS). [12]

Because scalability affects the availability of a web service it is important for the owning business. All of the downtime of a web application can be counted as a loss of potential revenue. The size of the impact is based on the intended scope of the service itself. If the application in question is a simple webshop for a local business, downtimes during the night will hardly have an impact. In contrast, downtime in a service such as Netflix is avoided because of the scale of the impact on the service.

Robustness is a measure of how well a software handles changes to the existing itself, the hardware infrastructure, or the external environment in which the application is running. A robust web application can adapt to these changes without breaking or malfunctioning and can continue to provide the desired functionality to its users. This is an important quality, as web applications are expected to be run in multiple web and mobile environments and must be able to accommodate changes in the environment without disrupting their operation. [13] On the other hand, a system low in robustness is a fragile one. Fragile software is such that a change has a cascading effect on the software and produces errors that have no clear causality connection to the change made [14]. As such robustness is an overarching quality that has implications for the other characteristics defined in this chapter.

Security of information in any software is one of the priorities, not excluding web applications. As most personal and official business can nowadays be conducted via the web, the security of the information and transactions on it has become a liability for the companies and governments that host their services online [7]. As software security is a rather large topic on its own, it will not be discussed in detail as a part of this work. In the EU area, the security requirements are mandated by the law, with the General Data Protection Regulation (GDPR) [15]. This has a ripple effect across most of the world as software that does not abide by these regulations cannot enter the EU market.

3 Modularity and change

Enabling change is a challenge in any software project. Many software quality attributes have trade-offs, which is true for change-enabling factors as well. The challenge is finding a balance that brings the most advantage and the least disadvantage considering the purpose of the software being developed.

Modularization is the process of breaking a software system into preferably small, independent modules. Each module typically focuses on a specific aspect of the overall system and has an interface that allows it to interact with the other modules. This makes it easier to comprehend, maintain, and modify the software, since changes to one module can be made without affecting the rest of the system. [16]

Modularization has been described as enabling software evolution. It allows for greater freedom in future design and implementation of changes by increasing the extent to which a system can be reorganized, and existing parts recycled. [16] A module is a structural component that contains similar functionality or shared responsibility [1]. It can either have only one purpose or be a collection of utilities with similar purposes. This has many benefits such as clear boundaries to make the structure easier to understand because of the reduced complexity [16], and it decreases duplication of code. Ideally, the responsibilities between the components in a module should be separated by role and context [1]. A properly modularized software system is such that a module with its dependencies can be extracted from the system to be reused in different contexts. [16]

Introduced in the following paragraphs, are some of the software architecture quality attributes that affect the modularity of software [17]. In the relevant literature these factors have also been described, to a degree, to exist independently, and not necessarily as part of modularization [1]. However, these enabling techniques overlap, and they do not necessarily exist independently from one another. As each of them concerns the division and definition through the separation of concerns in software architecture, they will be discussed as the enablers of modularity in this work.

Abstraction is a technique that combats complexity in software architecture by making the code and structure more generalized in nature and not case-based. It hides unnecessary details of implementation from the developer in a way that makes a full understanding of its functioning redundant while also increasing understanding of the operation by making it more straightforward. For example, separating the interface from the implementation of a component makes the use of the component simpler, because it hides all the unnecessary details. [1] Analogously, the details of how a car works may and should remain hidden from the driver [17]. All the driver needs to know is how to operate the car. How it works may be entirely irrelevant to the task at hand [18].

Similarly, **information hiding** and encapsulation are about hiding information and limiting access. Information hiding is about limiting access on the class or object level. Hidden information can be only accessed from inside the class or the object itself. It means controlling the internal state of an object or class by controlling data access [1] with for example, the commonly used *get-* and *set-* functions and *private* property modifiers.

Encapsulation is closely related to information hiding as it, concerns wrapping data and functionality in a structure, such as a class. Encapsulation is a useful tool when striving for cohesion as clear separation makes it easier to group related data and functionality. [1]

Loose coupling and **cohesion** are two related concepts that are important for the design of modular, maintainable systems. Coupling means the strength of the bonds between modules. A loosely coupled structure promotes little to no dependency in inter-module relationships. Cohesion, on the other hand, is a measure of the conceptual and functional connectedness of the structure the components of a module form together. Several different types of cohesion can exist within a software module, including functional, logical, temporal, communicational, and procedural cohesion. These types of cohesion describe how the elements within a module are related and work together to perform specific functions or processes. In a module, functional cohesion should be strived toward when possible, as it ensures all the elements in a module are working for one purpose. This makes the module easier to understand and modify. [1]

Following the **sufficiency**, **completeness**, and **primitiveness** technique, each component should be easy to interact with and be complete in its relevant characteristics

[1]. Sufficiency, completeness, and primitiveness are all qualities that can be difficult to measure in a software architecture. One way to measure sufficiency is to determine whether the component has all the necessary functionality to fulfill its intended purpose. This way the resulting interface would be minimalistic and concise. Completeness can be assessed by evaluating whether the component includes all the necessary details and features in its interface to perform its intended functions. Primitiveness can be determined by evaluating whether the component is built using fundamental and reusable constructs defined by the programming language. [19]

Divide and conquer is a top-down approach to designing software, dividing it into smaller parts that can be designed and implemented independently [1]. It is a technique used in software architecture to break down a complex problem into smaller, more manageable substructures with succinct interfaces. This way each substructure can be worked on independently, and together forming the overall structure of the software. This approach can make it easier to design and implement complex systems, as well as make the resulting system more efficient and scalable. [20]

4 Microservices architectural pattern

The web application architecture should be chosen to reflect the requirements for the software. There are various commonly recognized architectural patterns for software architecture. In the following list by Richards [21] (with the inclusion of service-oriented architecture) are short introductions to the ones most commonly implemented to emphasise the properties of microservices architcture in this chapter.

1. **Layered architecture** consists of layers with separated purposes. These layers form logically coherent units with the separation of concerns between layers. Even though layering, to a degree, may be used with other patterns, an architecture is considered to be layered if the layers alone form the basis for the structure and there is no further architectural division. A software architecture may implement as many layers as preferred. As an example, a layered architecture could include at least the following layers: *presentation*, *business*, and *database* layers. The presentation layer would include the user-interface and browser communication logic while the business layer would do calculation or data aggregation. Database layer would do all of the database interaction.
2. **Microkernel architecture** uses a single component as a basis for the whole system. The system in itself contains a minimal amount of functionality, but contains everything necessary to be functional on its own. The core system may be extended with stand-alone plugin modules that contain specialised logic. The microservices architecture can only be implemented in simple and small scale systems as the complexity of the system will increase rapidly with the growth of the core component.

3. **Space-based architecture** uses a tuple space implementation of processing units and virtual middleware to create systems that are scalable on-demand. The virtual middleware includes components such as a messaging grid, a data grid, a processing grid, and a deployment manager. With these components, the middleware takes care of the communication and task delegation for and between the processing units. A space-based architecture has no central database. Instead, each replicated processing unit contains in-memory data that is updated by a data replication engine, a part of the virtual midlleware.
4. **Event-driven architecture** emphasises the importance of events in the design of the architecture. There exist two distinct types of event-driven architectures, namely the *mediator* and *broker* topologies. Using the broker topology, the logical components that form the system are decoupled as they simply react to events. The information about an event is passed to an event-broker which passes them on to relevant subsystems. The mediator topology makes use of an event mediator that receives the events, passing them on to relevant subsystems for further processing.
5. **Service-oriented architecture** might seem similar to layered architecture in the regard that it has relatively sizeable service components, driving the separation of concerns in the application. However, services are autonomous and reusable components, that are remotely called. None of these properties are necessary for a layered architecture. [22]

Out of the ones mentioned, the last four are the most suitable choices for a web application. [21] As with other types of software, web applications have their own set of requirements by default. Web applications run in a browser on a desktop or mobile environment, which already imposes some requirements on the software such as cross-platform support for example. It also means that not all architectural patterns are equally fit to serve the requirements of web applications. In this chapter and the rest of the work,

the focus will be on microservices architecture and the implications of implementing it in a web application.

Software architecture should be impartial to the implementation. It should describe the structure of a system on an abstract level. Not the actual implementation, including the choice of programming languages, frameworks, databases, and platforms. It is common to design architectures around technologies but following a technology-agnostic design (TAD) and technology-agnostic architecture (TAA) will decrease risk and increase the scalability and availability of the application. [23]

Software architecture affects how well software will stand up to time. It is a key factor when designing and implementing software because it influences the whole journey. A poorly defined or implemented architecture might result in software that will be increasingly hard to understand and maintain over time, as changes are made, and requirements change. Different architectural patterns can be used, each having its strengths and weaknesses. To achieve the best results, the requirements should be carefully observed while planning the architecture. [24]

Web application architecture can be divided into groups by functionality. This is most often presented as the presentation, service, and data access layers, each having its own functionality. However, these are conceptual definitions, meaning that they do guide the structure, but they do not mandate a specifically defined division into any specific sub-structures. [25]

In contrast, an architectural pattern mandates the architectural structure to be designed and implemented in a certain manner. This clarification is intended to make the distinction between an architectural pattern and the modeling of layers of the architecture.

4.1 Microservices

Microservices architecture is one of the most used architectural patterns in web applications. As the name suggests, a ***micro***service cannot be as big as the application itself. Instead, it is a functional sub-system that can stand and execute on its own. Ideally, these microservices are loosely coupled, meaning that in the perfect scenario, they could be turned on and off with few or no side effects [26].

Microservices architecture could be thought of as a second iteration of the service-oriented architecture pattern [27]. The service-oriented architecture pattern was developed to answer the need to have abstract service components with as little dependency on each other as possible. By reducing mutual dependency between areas of the architecture, greater flexibility was achieved in the extent to which the software could be modified. [22]

Service-oriented architecture has various advantages. It enables dynamic software where multiple instances of the same service can be launched to split the service load. Heightened modularity and reusability also come with the encapsulating of the architecture to service components, enabling a high level of cohesion and enabling loose coupling. This in turn promotes distributed development as it makes parallel development easier. The possibility of integration into other systems, not excluding legacy systems is also an advantage. It even makes gradual migration possible. [27] However, what service-oriented architecture lacks in comparison to microservices architecture is that it does not consider the size and autonomy of the service components in the same way that microservices do [22].

Microservices architecture isolates software degradation to the service component level, making the software much easier to repair and maintain [28]. Implementing a microservices architecture in a web application will have a change-enabling effect on the software. Because many web applications of today need to be accessible continuously, it is more important than ever to have as little maintenance time as possible. This is to reduce the loss of revenue incurred during the maintenance break. In a web application with microservices architecture, it is possible to update components individually without bringing the whole system to a halt.

Figure 1 from Mark Richards' book, *Software Architecture Patterns*, pictures the microservices architectural pattern giving a basic example of the structure and

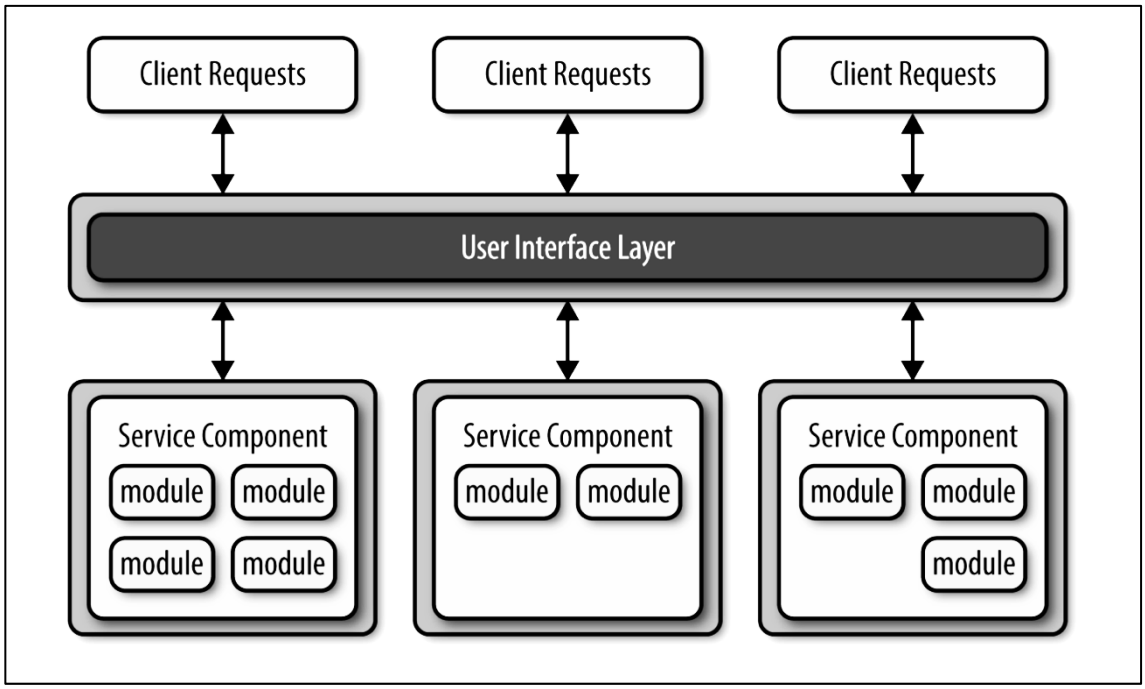

Figure 1 – Microservices architectural pattern [21].

communication. It consists of decoupled, *service components* that each consist of one or more modules. With microservices, client requests are usually handled by a frontend *user interface layer*, which communicates with individual service components that may communicate with a database. A set of similar service components that share a purpose should ideally also have their own databases to promote the decoupling of information on the database level as well. Together the service components form the application. Individuality enables scalability as new instances of a service component can be created to handle each incoming request. [21]

Each service component is ideally designed to handle a specific set of tasks or responsibilities. They should be further divided into smaller units called *modules*. A module should be kept small and, as the service component it is in, focused on a specific responsibility or a task. [21] Ideally, this would result in a cohesive structure where each module is unified logically, sequentially, communicationally, or procedurally.

Mark Richards' analysis of microservices architecture shows that it is high in *agility*, *ease of deployment*, *testability*, *scalability,* and *ease of development*. All of these

are largely the consequence of the independent nature of the service components as it makes all manner of changes much easier to implement. But it is also because of its distributed nature, that microservices architecture is probably not the best solution when high performance is among the requirements. [21]

The need to transfer data between service components in a microservices application increases its vulnerability and the risk of being exposed to security threats. This sets requirements for overhead in encryption and other countermeasures as early as the design phase of a microservices application to ensure security. [27] Also, a study on software architecture security pointed out that two instances of a service component should not be running locally on the same system. This is because it may make it more difficult to track which instance is responsible for handling a particular request or to perform updates and other maintenance on the system. [29]

It is comparatively easy to implement changes to service components due to their loosely coupled nature and when they are kept relatively compact. [27] The small size makes service components easier for developers to understand potentially increasing the understandability of the whole software and enabling change.

Microservices architecture potentially offers a good environment for testing as individual service components can be tested in isolation on the unit level and communication with other service components can be easily simulated in tests. However, as a tradeoff, this might come with increased complexity when it comes to integration testing of the application when testing very large systems that have very many service components. [27]

5 TypeScript

TypeScript is a popular programming language when it comes to web application development. It might be even the most sought-after by some recent accounts in the media [30, 31]. The purpose of TypeScript is related to enabling changeability in areas where JavaScript falls short of the goal, such as scalability and ease of maintenance [32].

TypeScript was developed by Microsoft in 2012. It was developed in such a way that it encapsulates *JavaScript* wholly. All JavaScript code is by definition valid TypeScript as well, but not vice versa as JavaScript is a subset of TypeScript. At compile, time TypeScript code is first transpiled into JavaScript and then compiled as JavaScript into bytecode. Because TypeScript is compiled into JavaScript this also means that it can be hosted on any browser and platform that supports JavaScript [32]. This is an advantage as implementing a new programming language in an existing project would be a much more time-consuming errand than beginning to use TypeScript instead of JavaScript, which is more like plug-and-play. It is also easy to start using TypeScript incrementally as it would work very well with an existing JavaScript codebase.

The defining difference between TypeScript and JavaScript is that JavaScript is a dynamically typed language that can be used in an object-oriented fashion with prototype objects. In contrast, TypeScript adds class-based object-oriented programming and static types. With class-based object-oriented programming TypeScript allows drawing clear boundaries in code and data, that can be wrapped inside objects that are based on classes. Static types are advantageous for a variety of reasons. With TypeScript, the code will be type-checked at compile time before it is translated into JavaScript. This means that TypeScript enforces type safety at compile time to minimize errors and bugs that the dynamically typed JavaScript will not do. With dynamically typed languages, the type checking happens at runtime – not at compile time. This can lead to weird runtime behavior that can be much harder to troubleshoot because at runtime the program can't be expected to crash every time something unexpected happens with the typing. [32]

As opposed to JavaScript, TypeScript does not do Type Coercion. This means that when comparing variables with TypeScript, they must be of the same type for the code to compile. However, TypeScript also does type inference when concatenating values together. This might also lead to unexpected behavior from the program. However, TypeScript has been designed to work in unison with certain IDEs such as VSCode, making this a minor problem. [32] This is important as this is the lowest level at which a software system's properties may be influenced. Not even the code itself but the process of writing it down.

JavaScript was never designed to be used in large applications and is best suited for small-scale use in relatively simple scripts. TypeScript on the other hand extends the possibilities of JavaScript to the large-scale application development level. [32] The reason why extending JavaScript is very useful is simply put that JavaScript has achieved universal browser support [33]. In addition, in a statistics report by W3Techs, it is stated that "JavaScript is used as a client-side programming language by 97.7% of all the websites" [34]. Because portability is a requirement in modern web applications [13], using JavaScript is a good choice when implementing a web application. By extension, TypeScript inherits JavaScripts advantages while it also counters some of its disadvantages.

TypeScript employs two different approaches to type checking, structural typing, and static typing. Structural typing is a form of type compatibility that is based on the structure of types, rather than their names. This means that two types are considered compatible if they have the same properties and methods, regardless of whether they are part of the same type hierarchy or not. On the other hand, for a language to be statically typed, the types must be known at compile time. This means that the type of a variable or expression must be known and fixed when the code is written and cannot be changed during execution. In TypeScript, types are defined using interfaces, which specify the structure of an object. This allows TypeScript to check the compatibility of objects at compile time, ensuring that they have the correct structure and can be safely used together. The types are used through type annotations when declaring functions, variables, and functions. TypeScript's support for structural typing allows the creation of modular code that is easily extensible and can be changed and adapted over time. [35]

According to Bruce [36] type systems are usually designed to enable qualities such as *runtime safety*, *optimization*, *documentation*, and *abstraction*. As types provide the compiler or interpreter with useful information, it can help the compiler or interpreter to better optimize the program. This also reduces the need for type checking in the code because type checking is done by the compiler or interpreter that throws type errors if a typing mistake is found. This is good as it doesn't allow the execution of this kind of illegal statement at all. [36] A study was done in 2012 that compared Groovy, a dynamically typed programming language to Java, a statically typed programming language. It was an empirical study from the point of view of the maintainability of software systems. In the study, Kleinschmager et al. [37] found that static typing helps developers use *new* classes and makes fixing type errors easier.

As TypeScript employs a type system with static types, the code itself is self-documenting to an extent as the type provides valuable information to the developer when studying or writing code. However, a statically typed language by itself is not a substitute for documentation. Having to annotate the constructs in code with types, such as variables and functions return types, adds a self-documenting layer to the code. Being able to define types and hide information inside of types also allows for a greater degree of abstraction of the code. [36] Modern IDEs offer useful functionality regarding the analysis of the code as it is being written. They can do the type-checking without compiling the code. This is highly useful as it reduces the number of times that a piece of code must be compiled and run while writing it and makes spotting type errors faster. There is also the added benefit that the IDE can give the developer suggestions on what operations can be done to a construct based on its type.

The type system is also structurally typed. Structural typing concerns the relation of types based on their structure rather than the name of the type. This is called type compatibility and means that two types can be compatible even though one does not declare itself as implementing the other. Structural typing allows greater flexibility with the use of types as a type does not have to be equal in structure, reducing boilerplate code. [35] Boilerplate code refers to code that must be repeated in many places perpetually with little or no change in its structure. The use of generics may also result in unsafe operations at runtime [38]. An example of the type compatibility that structural typing enables is illustrated in Figure 2, an example from the official documentation of TypeScript [35].

```
interface Pet {
  name: string;
}
class Dog {
  name: string;
}
let pet: Pet;
// OK, because of structural typing
pet = new Dog();
```

Figure 2 – Type compatibility in TypeScript [35].

However, even though the type system is designed to be as sound as possible, there are some unsound operations. A sound type system is such that the operations that are being executed at runtime are known to be safe. An unsound operation is demonstrated

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;
y = x; // OK
x = y; // Error
```

Figure 3 – Comparing functions in TypeScript [35].

in Figure 3. As *x* must have all arguments of corresponding types to *y*, *y* = *x*, commented with "OK" is valid. If the *x* = *y* on the following line was run instead of *y* = *x*, an error will occur. As *x* does not have the property *s*, *x* = *y* is invalid, but accepted by the compiler. This is intentional as it ensures backward compatibility with JavaScript implementations. [35] The drawbacks to unsoundness include limiting the effectiveness of type annotations and making abstraction harder while also even requiring type checking in code in some situations. [38]

Another source of unsoundness in TypeScript is the concept of *any type*, that accepts literally any type of value as input. As this also is a special case accepted by the TypeScript interpreter, it is a possible source of unsound behavior. *Any type* is treated like it could be of any type at runtime, which can lead to unsound behaviour as it circumvents the type checking done by the compiler. [39] In contrast, TypeScript employs generics, that reduce the duplication of code by allowing arguments of a generic type. For example, a function that takes an argument of a generic type can be given an argument of any

```
function identity<Type>(arg: Type): Type {
  return arg;
}
```

Figure 4 – A generic funtion [39].

available type. Although the use cases for generics are somewhat limited because using them may result in more boilerplate code with the addition of type constructors as well as possibly increasing the complexity of the codebase as perceived by the developer. The example in Figure 4, showcases a generic function. In the example, the function called “identity“ may take arguments of Type, which means that any type of argument can be given to the function. The TypeScript compiler treats generics as if they could really be any type. Which means that any logic that could not be true for each of the members of every type, can not be implemented. This makes the use of generics the type safe option instead of using an *any type*.

TypeScript also includes a set of primitive types, which are the basic building blocks of all types in the language. These include types such as number, string, and boolean, which represent numbers, strings, and booleans, respectively. These primitive types can be used on their own or combined with other types to create more complex types with interfaces and used with type annotations. [40]

In addition to the basic type constructs mentioned above, TypeScript also supports type extension and property modifiers. Type extension allows the creation of new types that are based on existing types and can include additional or override existing members. Property modifiers, i.e. public, private, and protected, can be used to control the visibility and accessibility of class members. [41]

6 Changeability

Modern web applications must have the ability to change. Rarely if ever a web application is considered to be complete at the launch of the first version. Even if all the requirements for it were satisfied at launch, the requirements are prone to change over time. At some point, new functionality might be needed or wanted. Or legislation might change. Or a better technology might emerge so that some aspect could be improved upon. Support for a framework or software library might end or some functionality might become deprecated or outdated. These are some examples of the possible drivers of change in software. Instead of avoiding change, great care should be taken to mitigate the negative effect called *technical debt* that follows change. This can be aided by considering the changeability of the software system at the initiation and as new changes are introduced.

6.1 Software erosion and technical debt

Technical debt is a term used to describe the cost associated with making shortcuts or compromises in the design or implementation of a software system. They may be relatively simple and fast to implement in the short term but can also make the system more difficult to maintain or update in the long term. As a result, technical debt can contribute to software erosion, and can make a software system less flexible and adaptable over time. In essence, technical debt incurs when a loan of time and convenience is taken against the quality of the technical implementation or design. Technical debt usually incurs from parts of the software that are not visible to the end user. This is because focusing on the visible part offers an immediate value increase to the software, making shortcuts and compromises more appealing. [17]

There are four types of technical debt, and they are implementation debt, design debt, test debt, and documentation debt. Implementation debt can be observed through

code smells. Common code smells are, for example, long or non-descriptive method names and the duplication of code. Design debt, on the other hand, incurs when a design decision is executed that does not adhere to the boundaries set for the architecture. [14] For example, if a service component in a microservices' service component is implemented in such a way that it is not decoupled or loosely coupled. This would result in a service component that is dependent on another component to function, making it not autonomous and this way distancing it from the ideal of the architectural pattern.

Design and architectural debt manifest in structural smells such as too complex structures or divergence from the planned architecture. Test debt is the consequence of low test automation coverage, and otherwise missing or inadequate testing. Documentation debt entails inadequate, outdated, or altogether missing documentation concerning the implementation. However, only documenting the implementation will not be enough, to not result in any documentation debt. Also, the architectural design and decisions made should be documented. Elsewise valuable knowledge of the process will be lost making change harder. [14]

The process of incurring technical debt is called *software erosion*. Software erosion happens over time, every time a change is made to the software where debt is taken. This way the software system gradually becomes more difficult to maintain or

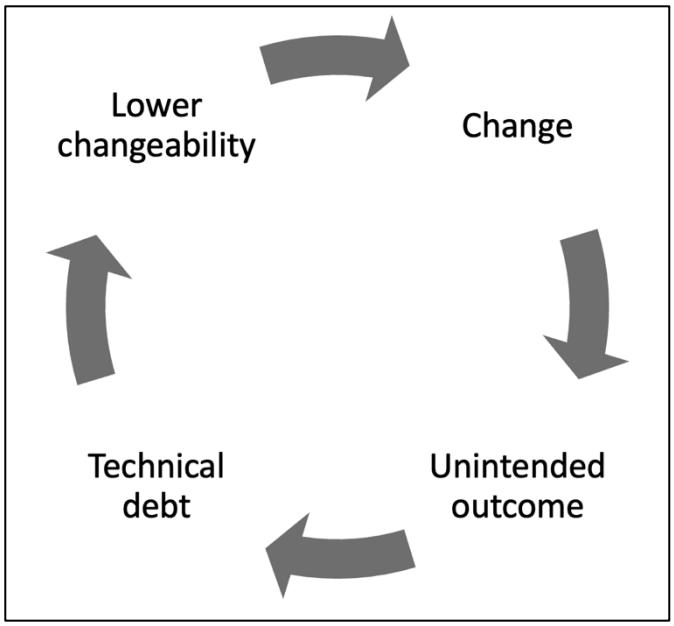

Figure 5 – The self-perpetuating nature of software architecture erosion.

update over time. [5] As software erodes it incurs technical debt. The erosion is the result of the lack of changeability. The cycle of software erosion has been illustrated in Figure 5. It is a self-perpetuating cycle. Not all changes cause erosion but the more it has happened, the more likely it is to happen again. Over time, software erosion can make a software system less flexible and adaptable, making it increasingly difficult to implement changes or updates to the system.

In the 1970s, among the first descriptions of software evolution were in Lehman's [4] work. They have become to be know as *Lehman's laws* of software evolution. They are a collection of key principles that dictate the course of software evolution [4]. There are a total of eight Lehman's laws. Out of the eight, laws 1, 2, 6, 7, and 8 are listed below as they describe the self perpetuating nature of software erosion [4].

L1 *Continuing change* – A software has to be continually changed to remain appelaing to its users.

L2 *Increasing complexity* – A software will grow more complex by time unless counter measures are taken to prevent it.

L6 *Continuing growth* – A software will have an increasing amount of functionality to remain appealing to its users.

L7 *Declining quality* – A software will decline in quality unless it is adapted to changes.

L8 *Feedback System* – A software's evolution is dependent on the succesfull implementation of changes on the system. Growing complexity and declining quality make further changes increasingly difficult and time consuming, inducing a self-perpetuating feedback cycle.

The changeability phenomenon will not be discussed through the Lehman's laws in this work. However, they are worth mentioning in this context as they due to their relevance in this subject area.

6.2 Changeability

As time goes on and software systems get ever more complex, it becomes increasingly important to design systems in a way that they maintain or even increase their lifecycle

value. This is an especially important topic regarding so-called *enduring systems* that are expected to have a long life because long life increases the impact of software erosion in a compounding fashion over time. [42] One way that this can be aided is by designing systems for change. This means designing a software system's architecture in a way that enables change during the system's lifecycle. [13] The extent to which a system can be changed is called **changeability** [1].

In simple terms, changeability is about the aging of software. As an analogy, as healthy lifestyle choices aim at slowing down the effects of aging in humans, changeability is about all the factors that influence longevity in software. Suboptimal solutions concerning changeability will lead to reduced life expectancy for software. Software aging has been talked about since at least 1994 when Parnas named the concept [6]. Not long after in 1996 Buschman et al. were among the first to talk about changeability in the software lifespan context [1].

Changeability is a non-functional property of software architecture. To illustrate what it means to be a *non-functional property*, looking into functional properties first might help to drive the point home. A *functional property* of software describes how the software or its component will behave given certain inputs and outputs. Putting it more simply, a functional property is "*what it does"*. A non-functional property of software architecture on the other hand is emergent, answering the question of "*how it is*". [11] It could be described as a trait the software has. Following this logic, functional properties could be better described as features. A non-functional property stems from the functional properties and can to a certain extent be extrapolated from them but does not necessarily have a straightforward causality connection to the functional properties. For example, it's like seeing white on a computer monitor. All that is happening on the monitor is that there are red, green, and blue (RGB) light sources emitting light with even intensities. Despite that, it is perceived as white. So, describing **how** it is, we can say the monitor projecting white light. When describing **what** it does, we can say the RGB- light sources are emitting light at the same intensities. Because of the ambiguous character of non-functional properties, it is harder to get meaningful results while measuring them making changeability and other non-functional properties an interesting but challenging subject of study.

As non-functional properties of software architecture are as important as functional properties, they are as important to study as well [1]. Thinking about the functional properties comes naturally as functionality is a natural consequence of the system that is being designed and implemented. Functional properties have a distinct causality, may be traced from the structure, and are easily understood by developers, designers, and architects. This is not the case with non-functional properties.

There are other ways to describe the phenomena relating to changeability. Buschman et al. were among the first to talk about changeability in their book *Pattern Oriented Architecture* from 1996, changeability can be divided into four categories. They are called **maintainability**, **extensibility**, **restructuring**, and **portability** [1]. They will be introduced in the following Section 6.3. After that, the enablers of changeability will be introduced concerning these four changeability aspects in Section 2.2.

6.3 Changeability aspects

Maintainability refers to the ease with which the software can be maintained. Without a properly maintainable architecture and codebase, the software will mature at a much faster rate compared to the contrary. One survey found that up to 90% of software costs are incurred during maintenance [43]. This would mean that increasing the maintainability of the software would decrease maintenance time and thus the cost of software in an almost linear fashion. By the definition given by Buschman et al. [1], maintainability refers to repairing the system after errors occur. To mitigate the effects of errors, it is beneficial to arrange the software architecture so that side effects outside individual components in the system are minimized. [1] The term "component" is used freely here to describe a substructure of software and is not based on any particular definition of a *component*. On the software architecture level, this could mean, for example, the utilization of a microservice architectural pattern in which the software's architecture would be divided into independent components to reduce mutual dependence. Maintainability can be seen as a key enabler for achieving good security, as it allows for timely patches and updates to be applied to the application, which can help protect against potential vulnerabilities or attacks.

It should be noted that maintainability is a commonly used word in software development, for example with ISO standards [44], and what was described here in this chapter, refers solely to the particular aspect of changeability that was introduced [1]. This is an important distinction to make as maintainability in itself could be used as a blanket term in casual conversation referring to any aspect of changeability. However, in this work, maintainability relies on the definition given in this chapter.

Extensibility refers to the ability to add new features or capabilities to the software without affecting its existing behavior. An extensible software is such that new features or functionality can be added to it without significant changes to the underlying code or architecture. This can include adding new modules, functions, or interfaces, as well as extending the existing functionality of the software. Extensibility is important because it allows the software to evolve and adapt to changing business requirements or user needs. For software to be extensible it must implement loose coupling in its components. [1]

Coupling between components and modules increases the difficulty of implementing changes to the architecture. A modification to a component mandates modification to it's coupled components as well, making it less trivial to extend. Coupling to an extreme degree will make a system increasingly fragile with time as every new feature will be harder to implement. Loosely coupled components depend on one another as little as possible to increase flexibility in the use of existing components and the addition of new ones. A system that has loose coupling will have a clearly defined structure and a clear division of the functionality of the software. [45] As loosely coupled systems have loose ties between components this also makes it easy to introduce new components or even entirely remove existing components with minimal effect on the functioning of other components.

An extensible application may be easier to keep available while minimizing the impact of service breaks due to their reduced length. Loose coupling between modules and components is beneficial in achieving extensibility because this way new components and modules can even be taken into production without the need for a service break. In addition, this way updates could be done on one component at a time without the need for application-wide service break, because the modularity of the system would allow for servicing the modules individually. Similarly, new components could be introduced without the need for a service break.

Restructurability is the extent to which a system's components can be reorganized, even redistributing them between subsystems [1]. Because improvements are probable and because there might come a time when the software needs major revision, flexibility in this regard is important. There exist many factors that negatively influence the restructurability of software such as lack of good documentation, code that does not conform to standards, lack of code formatting, inhouse solutions in place of packages and frameworks, and lack of modularity. However, restructurability is not to be confused with optimization. The difference is that restructurability is about the comprehension of the code, structure, and architecture as well as readability while optimization concerns performance. A restructurable software would be such that it has good documentation, and test coverage, takes complexity issues into account, and is easy to comprehend. [46]

The ISO/IEC 25010 standard defines **portability** as the "degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another" [44]. For a system to be portable it must be able to be implemented in different environments from hardware platforms to operating systems and user interfaces, not being tied to any specific infrastructure. Portability is important because it is expected that a web application should run on the most prevalent browsers on desktop computers as well as in mobile browsers. It also must be structured in a way that enables the use of different programming languages and compilers in tandem. [1]

Portability has also been called "robustness" [13] as described in Chapter 2. A system is robust when it is insensitive to changes in the environment, meaning that it is expected to be able to function normally under different operating conditions [13]. As an analogy, you would expect to be able to drive a car whether it's sunny or raining and whether the road is paved or gravel. In each of those conditions, you would still expect the car to function normally and so it does. The car is a robust system under those conditions. Portability and robustness both describe the same property, but from different perspectives. A portable web application may be more scalable, because it may be easier to move the application to a different environment or platform that is better suited to its needs.

7 Enabling changeability in TypeScript web applications with microservices architecture

Modularization influences the changeability of software and there is a multitude of ways that it can be accounted for in software [1, 17, 18]. In this chapter, the changeability of web applications is examined through the enabling effect that the microservices architectural pattern and TypeScript have on the modularity of a web application. The architectural pattern used limits the possibilities of implementing the structure of the application as does the choice of the programming language it's written in.

Earlier in this work, modularization was introduced as a change-enabling factor of software architecture. In this chapter, the impact of modularization will be discussed through how TypeScript and microservices architecture influence their implementation.

7.1 Changeability in web applications

Following is a summary of the relation of the three characteristics of modern web applications, scalability, robustness, and security to the changeability aspects that were discussed. It emphasizes the relationship between the different web application requirements to the different changeability aspects. On the other hand, it also demonstrates the importance of design for change as one change can have an impact on more than one functional property of the software, but it can also simultaneously have an impact on more than one non-functional property. This also demonstrates how software systems can quickly become very complex as changes compound over time.

A **maintainable** web application will have clear and well-documented code, making it easier for developers to understand and modify the application as needed. This can help to ensure that the application remains robust and secure, even as it is updated with new features or fixes.

An **extensible** web application will have a modular design, allowing new components to be easily added without disrupting the existing structure of the application. This can help to improve the scalability of the application, as it can more easily accommodate new features and functionality without requiring significant changes to the underlying codebase.

A web application that is **restructurable** should be able to easily modify or reorganize its internal structure without affecting its overall performance or stability. This can help the application to scale better, as it can more easily accommodate changes in workload or traffic.

Portability has also been called robustness. A **portable** web application will be designed in such a way that it can be easily adapted to run on different operating systems or hardware, without requiring significant changes to the code. This can help to improve the security of the application, as it can be more easily deployed in a variety of different environments, making it more difficult for attackers to target a specific platform or configuration.

7.2 Changeability and microservices

The selection of an appropriate software architecture is crucial in the development of web applications, as it plays a significant role in determining the long-term viability and changeability of the software. Among the various architectural patterns available, microservices architecture is particularly well-suited to meet the unique requirements of web applications. [21]

In general, web application architectures consist of presentation, service, and data access layers that each has their own set of functionalities. However, the layers are conceptual definitions and do not dictate a specific sub-structure. [25] In contrast, architectural patterns mandate a specific architectural structure for the design and implementation of the software.

The microservices architectural pattern involves the creation of a software system as a collection of small, independent, and modular service components that can be modified and added to the system individually [21]. As this is a requirement for the

successful implementation of the pattern, it inherently supports the changeability of a software system. Because of this division, the pattern enhances the restructurability of the system and allows for extensibility. Additionally, because of the ability to replicate instances of service components to handle the increased load on the system [21], microservices enable scalability. The security concerns that come with scalability are in part countered by a distributed implementation where every instance is run in a different local environment [29]. However, it is worth noting that while microservices facilitate unit testing of small service components, they may also make integration testing more challenging if the application consists of a large number of different service components [27].

Microservices architecture enables the maintainability of software, requiring it to be split into service components that enforce a modular approach to the software's design. This helps trace errors back to their origin in a service component and through log entries of the communication between modules. It isolates changes at the service component level reducing the impact on the architectural-, and system-wide levels. This is also how the microservices pattern facilitates the use of the divide and conquer technique as its service components are on the highest level of the architecture.

The pattern facilitates portability as service components can be developed and deployed independently, making the migration to a different environment or platform easier.

Table 1 emphasises the relation of the discussed attributes of microservices to the modularity principles as discussed in this Section 7.2.

Change enablers	Microservices
Abstraction	
Information hiding	
Encapsulation	X
Loose coupling	X
Cohesion	X
Sufficiency, completeness, primitiveness	
Divide and conquer	X

Table 1 – Microservices and modularity.

7.3 Modularity and TypeScript

TypeScript is designed to be a superset of JavaScript, adding a type system and class-based object-oriented programming to the language [32]. This can make it a useful tool for implementing a microservices architecture, as it can help to improve the changeability, and modularity of the codebase. As JavaScript has practically universal browser support [33]. By extension this makes TypeScript a compatible language in all web applications, making TypeScript as portable between browsers as JavaScript.

One of the key benefits of using TypeScript in a microservices architecture is its type system. The types offer a self-documenting layer to the code, although not replacing the need for good documentation [36]. The type system is statically typed, which can make it easier to maintain and extend the software over time, as changes to one part of the system are less likely to cause issues in others. This is because types of variables, functions, and other constructs can be specified in code, which can help prevent common type-related errors.

In addition to static typing, TypeScript also supports class-based object-oriented programming [32], which can improve the modularity of a microservices architecture. By defining classes, interfaces, and modules, reusable components can be created that can be easily combined and extended to create new service components. This can make it easier to manage the complexity of a microservices architecture, as each microservice can be implemented as a self-contained unit of code. The modular approach is further aided by the concept of modules that are defined on the file level. This helps keep the modules relatively small in size. Overall, the use of TypeScript in a microservices architecture can help to improve the maintainability, extensibility, restructurability, and portability, of the software while making it more scalable, as it offers many tools for supporting and enabling the modularity of the codebase.

TypeScript enables abstraction through its structurally and statically typed type system. Types allow the specification of an object, meaning that the structure of an object's properties and the types of their values can be defined. This way, a layer of abstraction can be created that allows an object to be considered through inputs and outputs rather than the implementational details while shifting the focus from low-level to high-level behavior. Also, generics may help with abstraction but may result in more

boilerplate, increasing the complexity of the code. Boilerplate code, on the other hand, may be reduced by type compatibility. However, due to the unsoundness, it may result in runtime errors in some cases as described before in Chapter 5.

Information hiding and encapsulation in TypeScript enabled by the use of primitive types, along with type annotations, interfaces, classes, and modules, can help to improve the modularity and maintainability of the code. By allowing the definition of new types and classes, these constructs make it easier to reuse and combine different parts of the codebase, and to make changes without breaking existing code.

Through class-based object-oriented programming, TypeScript allows for the definition of classes and interfaces that can be used to create reusable components as well as encapsulate code and data. This can help to improve the modularity of a codebase, as it allows developers to hide the implementation details of a component behind a clearly defined interface with clear boundaries. By adding static typing and class-based object-oriented programming, TypeScript can make it easier to manage the complexity of a microservices architecture, allowing developers to create distributed systems in a way that enables future change.

Table 2 summarizes the relation of the discussed attributes of TypeScript to the modularity principles as discussed in this section.

Change enablers	TypeScript
Abstraction	X
Information hiding	X
Encapsulation	X
Loose coupling	
Cohesion	
Sufficiency, completeness, primitiveness	X
Divide and conquer	

Table 2 – TypeScript and modularity.

8 Discussions

There has been an abundance of research concerning software erosion over the years. Changeability has been discussed with varying terminology and based on the literature review for this work there exists no consensus on which terms to use. The International Organization for Standardization i.e. ISO has defined a set of terms concerning change in software under the ISO/IEC/IEEE 14764:2022(en) standard [47]. The following list of terms is quoted from the ISO standards.

1. **Adaptive maintenance:** “modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment”
2. **Additive maintenance: “**modification of a software product performed after delivery to add functionality or features to enhance the usage of the product”
3. **Correction:** “change that addresses and implements problem resolutions to recover gaps and to make software operational enough to meet defined operational requirements”
4. **Corrective maintenance:** “modification of a software product performed after delivery to correct discovered problems”
5. **Emergency maintenance:** “unscheduled modification performed to temporarily keep a system operational, pending corrective maintenance”
6. **Enhancement:** “software change that addresses and implements a new requirement”

In the same ISO standard, the effectiveness and efficiency of making a change to software are called "maintainability" [47]. This is the same phenomenon that in this work is

referred to as changeability. In addition, there exist a wide variety of quality attributes for software as shown in Figure 6 [44]. As can be seen, from the literature referenced in this

(Sub)Characteristic
Functional suitability
Functional completeness
Functional correctness
Functional appropriateness
Performance efficiency
Time behaviour
Resource utilization
Capacity
Compatibility
Co-existence
Interoperability
Usability
Appropriateness recognizability
Learnability
Operability
User error protection
User interface aesthetics
Accessibility

Reliability
Maturity
Availability
Fault tolerance
Recoverability
Security
Confidentiality
Integrity
Non-repudiation
Accountability
Authenticity
Maintainability
Modularity
Reusability
Analysability
Modifiability
Testability
Portability
Adaptability
Installability
Replaceability

Figure 6 – Software quality attributes, ISO/IEC 25010 [44]

work and these standards, the issue of change, quality attributes, and other non-functional properties of software remain open when it comes to the use of terminology. Variance in terminology makes the study of change in software harder as the same term can be used to describe similar attributes of software, but not in an entirely equivalent context.

9 Conclusion

In this thesis, the changeability of a web application was examined through TypeScript and microservices architecture. Web applications require changeability to a high extent making it important to recognize what enables change in them and how. Modularization was introduced as one of the aspects that influence the changeability of a software system. It was discussed as an enabling technique for enabling change. Several factors related to modularization were identified, such as abstraction, encapsulation, and information hiding. Also, the significance of the architectural pattern in a web application was discussed and the microservices architectural pattern was introduced as a pattern that is suitable for web applications. After the introduction of web applications modularization, microservices pattern, and Typescript, changeability was established as a non-functional property of software, having four aspects: repairability, restructurability, extensibility, and portability.

The division of changeability to subcategories allows for greater insight into the actions that could be taken to combat software erosion and avoid technical debt. However, this area of research lacks a solidified basis for terminology. Through the literature review, it was found that there exist no concrete definitions for the changeability phenomena or its aspects and they are discussed with varying and overlapping terminology. Similarly, the factors concerning the enabling of change have vague definitions. This is mostly explained by it being a non-functional property of software. As non-functional properties are emergent and not concrete, a certain level of vagueness cannot be altogether avoided. Although it might be a hindrance when conducting research on the subject and searching for relevant literature.

It was found that the microservices architectural pattern has many change-enabling qualities in software. Among those qualities are loose coupling, a division into relatively compact service components, and further into modules. This separation enables changeability from the perspective of all the aspects that were discussed.

Through examination from the perspective of changeability, it was found that TypeScript has many properties that enable designing and implementing modular software. It enables abstraction, information hiding, and encapsulation of code by types, classes, interfaces, and modules. Because TypeScript encapsulates JavaScript it inherits the universal support of the language in the web environment. Coupled with the widespread use of JavaScript in web applications, it is likely that the language will remain supported for a long time which is beneficial as far as longevity is concerned.

In the end, software erosion may not be possible to fully avoid. Nonetheless, some measures can be taken to alleviate the negative impact of change on the high level through architectural design decisions and on the code level through implementational decisions and good practice. Understanding the drivers of change in modern web applications and their relation to the changeability aspects is a valuable asset in the design of a web application.

References

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. and Stal, M. 2009. *Pattern-oriented software architecture volume 1 : a system of patterns*. 1st edition. Chichester: John Wiley & Sons.

[2] Al-Salem, L. S. and Abu Samaha, A. 2007. Eliciting web application requirements – an industrial case study. *The Journal of systems and software* 80 (3), 294–313.

[3] Ginige, A. and Murugesan, S. 2001. Web engineering: An introduction. *IEEE multimedia* 8 (1), 14-18.

[4] de Silva, L. and Balasubramaniam, D. 2012. Controlling software architecture erosion: A survey. *The Journal of systems and software* 85 (1), 132-151.

[5] Lehman, M. M. 1980. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* 68 (9), 1060-1076.

[6] Parnas, D. 1994. Software aging. In: *Proceedings of the 16th international conference on Software engineering*, 279-287.

[7] Shklar, L. and Rosen, R. 2009. *Web application architecture: principles, protocols and practices*. 2nd ed. Place of publication not identified: Wiley.

[8] Emery, C. 2022. A brief history of web development. Read on 9.1.2023. https://www.techopedia.com/2/31579/networks/a-brief-history-of-web-development.

[9] Love, C. 2018. *Progressive web application development by example: develop fast, reliable and engaging user experiences for the web*. 1st edition. Birmingham: Packt Publishing.

[10] Haire, B., Henderson-Sellers, B. and Lowe, D. 2001. Supporting web development in the OPEN process: additional tasks. In: *25th Annual International Computer Software and Applications Conference*, 383-389.

[11] Georg, G., Aagedal, J. Ø., Mirandola, R., Ober, I., Petriu, D., Theilmann, W., Whittle, J. and Zschaler, S. 2006. Workshop on Models for Non-functional

Properties of Component-Based Software – NfC. In: *Lecture notes in computer science,* 210-216.

[12] Ruparelia, N. 2008. *Cloud computing*. Cambridge, Massachusetts: The MIT Press.

[13] Fricke, E. and Schulz, A. P. 2005. Design for changeability (DfC): Principles to enable changes in systems throughout their entire lifecycle. *Systems engineering* 8 (4), 342-359.

[14] Lilienthal, C. and Lilienthal, C. 2019. *Sustainable software architecture: analyze and reduce technical debt*. 1st edition. Heidelberg: dpunkt.verlag.

[15] GDPR.EU. General Data Protection Regulation (GDPR). Read on 17.1.2023. https://gdpr.eu/tag/gdpr/.

[16] van der Hoek, A. and Lopez, N. 2011. A design perspective on modularity. In: *Proceedings of the tenth international conference on aspect-oriented software development*, 265-280.

[17] Suryanarayana, G., Samarthyam. G. and Sharma, T. 2015. *Refactoring for software design smells : managing technical debt*. 1st edition. Waltham, Massachusetts: Morgan Kaufmann.

[18] Bjørner, D. 2006. *Software Engineering 1 Abstraction and Modelling*. 1st edition. Berlin, Heidelberg: Springer Berlin Heidelberg.

[19] Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J. and Houston, K. A. 2007. *Object-oriented analysis and design with applications*. 3rd edition. Place of publication not identified: Addison Wesley.

[20] Mens, T. and Wermelinger, M., 2002. Separation of concerns for software evolution. *Journal of software maintenance and evolution: research and practice* 14 (5), 311-315.

[21] Richards, M., 2015. *Software architecture patterns*. Vol. 4, 1005.Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Incorporated.

[22] Abbott, M.L. and Fisher, M.T., 2015. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Addison-Wesley Professional.

[23] Williams, B.J. and Carver, J.C. 2014. Examination of the software architecture change characterization scheme using three empirical studies. *Empirical Software Engineering* 19 (3), 419-464.

[24] Deinum, M. and Cosmina, I. 2021. *Pro Spring MVC with WebFlux : web development in Spring framework 5 and Spring Boot 2*. Second edition. Place of publication not identified: Apress.

[25] Digital Guide. 2020. Microservice architectures: more than the sum of their parts?. Read on 9.1.2023. https://www.ionos.com/digitalguide/websites/web-development/microservice-architecture/.

[26] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina L. 2017. 'Microservices: Yesterday, Today, and Tomorrow', in *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, 195–216.

[27] Shadija, D., Rezai, M. and Hill, R. 2017. Towards an understanding of microservices. In: *2017 23rd International Conference on Automation and Computing (ICAC)* 16.

[28] Ghiya, P. 2018. *TypeScript Microservices*. 1st edition. Packt Publishing.

[29] Jasser, S. 2019. Constraining the Implementation Through Architectural Security Rules: An Expert Study. In: *Product-Focused Software Process Improvement*, 203-219.

[30] Hale. C. 2022. JavaScript is no longer the favorite programming language for developers. Read on 9.1.2023. https://www.techradar.com/news/javascript-no-longer-the-favorite-language-among-developers.

[31] Devjobsscanner. 2022. Top 8 Most Demanded Programming Languages in 2022. Read on 9.1.2023. https://www.devjobsscanner.com/blog/top-8-most-demanded-languages-in-2022/.

[32] Maharry, D. 2013. *TypeScript Revealed*. 1st ed. 2013. Berkeley, CA: Apress.

[33] W3schools. JavaScript Versions. Read on 9.1.2023. https://www.w3schools.com/js/js_versions.asp.

[34] W3techs. 2023. Usage statistics of JavaScript as client-side programming language on websites. Read on 9.1.2023. https://w3techs.com/technologies/details/cp-javascript/.

[35] Typescriptlang. Type Compatibility. Read on 9.1.2023. https://www.typescriptlang.org/docs/handbook/type-compatibility.html.

[36] Bruce, K. B. 2002. *Foundations of Object-Oriented Languages: Types and Semantics*. Cambridge: MIT Press.

[37] Kleinschmager, S., Robbes, R., Stefik, A., Hanenberg, S. and Tanter, E. 2012. Do static type systems improve the maintainability of software systems? An empirical study. In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 153-162.

[38] Rastogi, A., Swamy, N. Fournet, C. Biermann, G. and Vekris, P. 2015. *Safe & Efficient Gradual Typing for TypeScript*. SIGPLAN notices 50 (1), 167-180.

[39] Moors, A., Piessens, F. and Odersky, M. 2008. Generics of a higher kind. *SIGPLAN notices* 43 (10), 423-438.

[40] Typescriptlang. Everyday types. Read on 9.1.2023. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html.

[41] Typescriptlang. Advanced types. Read on 9.1.2023. https://www.typescriptlang.org/docs/handbook/advanced-types.html.

[42] Browning, T. R. and Honour, E. C. 2008. Measuring the life-cycle value of enduring systems. *Systems engineering* 11 (3), 187-202.

[43] Chen, C., Alfayez, R., Srisopha, K., Boehm, B. and Shi, L. 2017. Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It?. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 377-378.

[44] ISO 25000. ISO/IEC 25010. Read on 9.1.2023. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[45] Pautasso, C. and Wilde, E. 2009. Why is the web loosely coupled?: a multi-faceted metric for service design. In: *Proceedings of the 18th international conference on world wide web*, 911-920.

[46] Arnold, R. 1989. Software restructuring. In: *Proceedings of the IEEE* 77 (4), 607-617.

[47] ISO. ISO/IEC/IEEE 14764:2022(en). Read on 9.1.2023. *https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:14764:ed-3:v1:en*.