

Nicolas Kivelä

DESIGNING ADAPTABLE WEB APPLICATIONS

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Joulukuu 2024

ABSTRACT

Nicolas Kivelä: Designing adaptable web applications
Bachelor thesis
Tampere University
Information technology
December 2024

The development of adaptable web applications is critical in today's rapidly evolving technological landscape. This thesis addresses the core challenges and solutions of designing adaptable web applications, which is highly valuable for businesses providing web application services.

The thesis discusses foundational concepts of web applications and the role of client-server models. It introduces the fundamentals of software architecture, focusing on quality attributes such as modifiability and adaptability. The research analyzes decision-making processes during the early stages of software projects, highlighting risk-driven approaches and modular design to limit costly refactoring.

The thesis introduces architectural patterns, from monolithic and layered designs to microservices architecture, which are examined for their adaptability characteristics. These patterns are compared in terms of their suitability for different phases of application lifecycle. The research also discusses low-level tactics, such as coupling reduction, cohesion enhancement, and module size optimization, to support adaptability.

The findings suggest that designing adaptable web applications requires understanding of the business situation and prioritizing the low-level tactics for modifiability. Monolithic architectures suit early-stage projects with small teams. However, providing modularization enables effective transition to more adaptable architectures, such as microservices. Preparing for changing requirements through risk-driven architectural decisions and modularization is essential for creating adaptable web applications.

Keywords: Software Architecture, Web Application, Adaptability, Modifiability

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Nicolas Kivelä: Adaptiivisten web-sovellusten suunnittelu
Kandidaatintutkielma
Tampereen yliopisto
Tietotekniikka
Joulukuu 2024

Adaptiivisten web-sovellusten kehittäminen on tärkeää nykypäivän nopeasti kehittyvässä teknologisessa ympäristössä. Tämä tutkielma käsittelee adaptiivisten ja muokattavien web-sovellusten suunnittelun keskeisiä haasteita ja ratkaisuja.

Tutkielmassa käsitellään web-sovellusten peruskäsitteitä ja asiakas-palvelin-arkkitehtuuria. Työssä esitellään ohjelmistoarkkitehtuurin perusteita keskittyen laatumääreisiin, kuten muokattavuuteen ja adaptiivisuuteen. Tutkimus analysoi päätöksentekoprosesseja ohjelmistoprojektien alkuvaiheessa, korostaen riskilähtöisiä lähestymistapoja ja modulaarista suunnittelua, jotta vältetään kalliit refaktoroimisprosessit.

Tutkielmassa esitellään arkkitehtuurimalleja, monoliittisista ja kerroksellisista rakenteista mikropalveluarkkitehtuureihin, sekä arvioidaan mallien muokattavuutta ja adaptiivisuutta. Näitä malleja tutkitaan myös sovelluskehityksen elinkaaren ja liiketoiminnan näkökulmasta. Tutkimuksessa käsitellään myös muokattavuutta edistäviä alatasen menetelmiä, kuten kytkentöjen vähentämistä, koheesion parantamista ja moduulien koon optimointia.

Tulokset osoittavat, että adaptiivisten web-sovellusten suunnittelu vaatii liiketoiminnan ymmärtämistä sekä alatasen muokattavuusmenetelmien priorisointia. Käsitellyistä malleista monoliittinen arkkitehtuuri sopii projektien ja yritystoiminnan varhaiseen vaiheeseen. Modulaarisuuden vahvistaminen puolestaan tehostaa siirtymistä mukautuvampiin arkkitehtuurimalleihin, kuten mikropalveluihin. Adaptiivisten ja muokattavien web-sovellusten suunnittelussa on tärkeää ottaa huomioon mahdolliset riskit ja varautua niihin käyttäen edellä mainittuja alatasen menetelmiä.

Avainsanat: Ohjelmistoarkkitehtuuri, web-sovellus, adaptiivisuus, muokattavuus

Tämän tutkielman alkuperäisyys on tarkistettu Turnitin OriginalityCheck -palvelun avulla.

THE USE OF AI TOOLS

AI applications have been used in my thesis:

No

Yes

According to my statement, I have used the following AI applications during the thesis process:

AI application names and versions:

Scopus AI

OpenAI ChatGPT GPT-4o

Purpose of use:

AI has been used for checking my grammar and finding synonyms for words (OpenAI ChatGPT).
Certain keywords for sources were discovered through the usage of AI (Scopus AI).

Sections where AI was used:

Abstract

1. Introduction
2. Web applications
3. Architecture and design patterns for web applications
4. Conclusion

I am aware that I am fully responsible for the entire content of my thesis, including the parts where AI has been utilized, and I accept responsibility for any potential violations of ethical guidelines.

PREFACE

Thank you for reading this paper!

Tampere, 20.12.2024

Nicolas Kivelä

SISÄLLYSLUETTELO

1. INTRODUCTION	1
2. WEB APPLICATIONS.....	2
3. SOFTWARE ARCHITECTURE.....	4
3.1 Software architecture in general	4
3.2 Quality Attributes and Requirements	5
3.3 Tactics.....	6
4. ARCHITECTURE AND DESIGN PATTERNS FOR WEB APPLICATIONS	9
4.1 MVC and Monolithic	9
4.2 Layered architecture.....	11
4.3 Microkernel.....	12
4.4 Microservices and Service-oriented Architecture	13
5. CONCLUSION.....	16
REFERENCES.....	18

ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
HTTP	HyperText Transfer Protocol
MVC	Model-View-Controller
URL	Uniform Resource Locator
SOA	Service Oriented Architecture

1. INTRODUCTION

The goal of this research is to gather and study the literature regarding software or web application designing to determine the main factors in designing adaptable and scalable web applications. Designing software is one of the core concepts in mastering the area of software development. It is the fundamental factor for successful, scalable, and adaptable software products. (Kassab et al., 2018)

The process of developing software from an idea to a product that creates value for the user is complex and there are many variables to consider. One of the most important but evolving aspects of the development process is the product requirements, which affect the business logic of the application. Finding and defining requirements is not straightforward but rather an iterative process and it evolves throughout the development project thus affecting the end-product and business logic. In theory, the modern software development process progresses through iterations, supporting adaptation to evolving requirements. However, in practice, does the software architecture and its implementation adapt to these changes? The maintenance phase of software, in which adaptation needs occur, accounts for a great proportion of the entire development costs. (Subramanian et al., 2001) How to prepare for changing requirements in a way that prevents additional costs and workload of redesigning and rewriting the applications? What are the factors in designing a solid but adaptable foundation for software applications?

This literature review includes material from Elsevier, ACM, and IEEE databases. Using search words: Web applications, Modular architecture, MVC, Software architecture, Microservices, Client-server, adaptability, evolvability, modifiability.

In this thesis, we will be discussing the fundamentals of web applications in Chapter 2. Chapter 3 provides information about the philosophy of software architecture and doing so with adaptability as a guide. Creating a basic understanding of the software architecture process and different aspects of it. Chapter 4 presents various architectural styles and patterns in evolutionary order. This chapter provides a brief comparison of their pros and cons regarding adaptability.

2. WEB APPLICATIONS

The Web as an environment has changed the way we access information and thus changed the software development field providing more opportunities for businesses to grow. Web applications are constantly evolving, and the competition is intense which forces rapid development of such systems. New technologies arise providing more opportunities for those who can adapt, on the flip side, taking companies out of business. To survive, in this quickly evolving field, it is necessary to implement new features to already existing systems.

The software that functions over the internet is called a web application (Hassan and Holt, 2002). This is the simplest definition for web applications but to function, these applications need a server, network, hypertext transfer protocol (HTTP) and a browser that serves as a client. In addition, client-server architecture provides the general structure of web applications. Web application's core characteristic is its ability to communicate with the user by providing dynamic data to the client. This is enabled by a database where the data is stored and can be fetched when needed. (Client-Server overview, 2024)

The connection between different modules such as client and server happens using HTTP (Conallen, 1999). As Tim Berners-Lee (1996) describes, "The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems." HTTP protocol provides the core functionality of the web and is based on the web application's core architecture, client-server. The client requests a specific path, or uniform resource locator (URL), with a method that defines the required action. Then the server receives the request and responds based on the path and method. The requested data, for instance, the HyperText Markup Language page is returned to the client (Client-Server overview, 2024) As demonstrated in Figure 1, how the components are connected with HTTP.

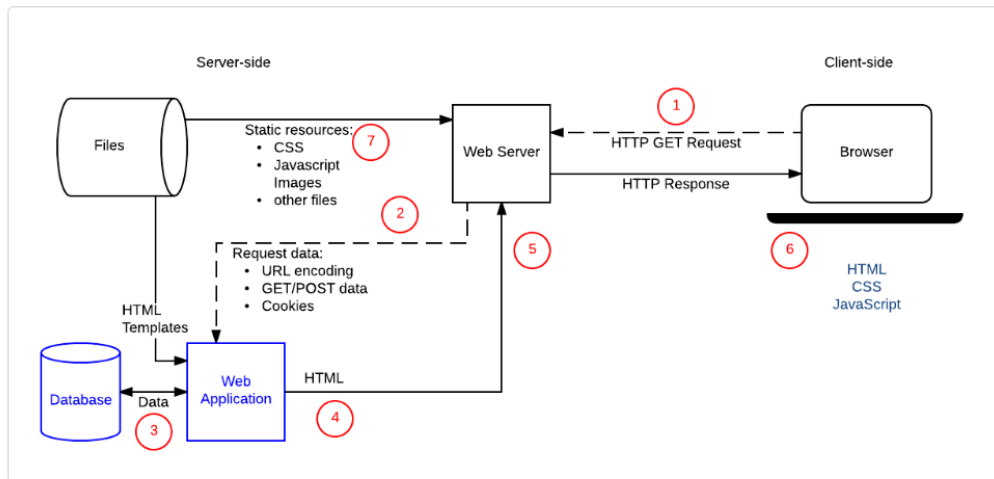


Figure 1: This diagram shows the main components of web application (Client-Server overview, 2024).

The client-server is a distributed structure where tasks are partitioned between service providers and requesters. The server functions as the provider and the client as the requester communicating over a network. (Architecture-client-server-model, 2024) It is the fundamental web application architecture pattern that physically separates the back and front ends of web applications.

The Web provides a platform that creates numerous web application business opportunities however the accessibility increases competition. Differentiating from the competition requires a deeper understanding of the requirements of the customer which again demands development iterations and adaptability from the architecture.

3. SOFTWARE ARCHITECTURE

Software architecture is a tool for partitioning to reason about the system's functionality. It provides a set of structures including software elements and their relations (Fairbanks, 2010, p. 16). These structures make it easier for developers to understand complex systems. Software architecture is used to design the wanted core structure of an application with needed quality attributes. In addition, these attributes are prioritized based on the business goals. As Bass (2012, p. 3) describes architecture connecting the business goals and the final application. In this chapter, we will be finding the fundamental aspects of architecture influencing the adaptability quality attribute of an application.

3.1 Software architecture in general

To understand which characteristics of architecture affect adaptability, we need to first understand why architecture is useful and important in software development. In addition to previous, architecture enables us to solve business problems by creating software solutions to match the quality attributes of the software with the available resources. Choosing the right architectural structure might help achieve, for instance, adaptability (Fairbanks, 2010, p. 18). There are multiple attributes that affect adaptability, one of them is modifiability which is affected by how the functionality of the system is divided and coupled. Systems modifiability can be measured by the number of separate elements that a specific change affects. (Bass et al., 2012, p. 27) Making the right architectural decisions to enhance these quality attributes, that support the software's adaption to changes, is the key to adaptability in software. However, making these right decisions is complex, and next, we will explore the decision-making.

Early phases in a software project are full of risks and unknown requirements, additionally, the early decisions are the most fundamental and they usually define how the rest of the project will succeed (Bass et al., 2012, p. 32). Therefore, it is important to understand the business goals and the different risks in order to make the right decisions. However, designing the whole architecture in advance is not usually the best choice because it is time-consuming, and every aspect of the system isn't known. Fairbanks (2010, p. 37) introduces a risk-driven approach that focuses on prioritizing the most important quality attributes of the software and using risk factors to guide the decision-making process. Additionally, Bass et al. (2012, p. 117) summarize – "One cannot plan a system for

all potential changes". They rationalize this by referring to the fact that planning too much will take time and accumulate overly costs. Overcoming this problem, they concur with Fairbanks (2010) that prioritizing decisions based on risks and their costs is essential. Designing the system as it grows seems to be inevitable, however there are decisions to be made in the early phases. If these decisions are bad or not discussed at all, it will lead to redesigning or refactoring of the architecture.

Refactoring, in software architecture, is described as the process of redesigning the system or parts of it. This process is expensive, signifying that there needs to be a major risk to justify it. Otherwise, if the need for refactoring is ignored, it will accumulate to the technical debt thus affecting the architecture adaptability. (Fairbanks, 2010, p. 58) One could think that, based on previous statements, refactoring is unpreventable on some level. Making architectural decisions based on the current risks and business needs seems reasonable, however what if the current risks and business needs change and the decisions are already made? Bass (2012, p. 27) concludes that software will need to change when adapting to new environments or integrating new features, not to mention these changes will require resources. There are three ways to determine possible changes: local, nonlocal and architectural. The local change affects only a single element, on the contrary nonlocal affects multiple elements however the architectural structure is not changed. The most difficult changes happen at the architectural level affecting the way elements interact thus system-wide modifications are needed. (Bass et al., 2012, p. 27)

If architecture is not designed for adaptability, occurring changes will require costly refactoring. For example, if a need for a new feature arises yet it requires changes at the architecture level, a few questions should be discussed before making the change. What is the risk of not implementing this feature? What will be the cost of the change in the architecture? At this stage, the software is not very adaptable to changes. To design adaptable architecture, one will have to consider what are the ways to minimize this architectural refactoring when new changes are made.

3.2 Quality Attributes and Requirements

We mentioned quality attributes briefly in the last section. In this chapter, the concept of quality attributes will be explained in more detail and practically. Quality attributes are measurable factors that can be used to test how well the system satisfies stakeholders' needs (Bass et al., 2012, p. 63). For instance, if the business goal is to fulfill the needs of multiple different user groups, it would be beneficial to use architecture tactics that

include modifiability to the system. This way it is more cost-effective to add new user groups to the system without having to redesign large parts of it.

Requirements define what the system needs to satisfy business needs. Quality attributes and requirements are interdependent, influencing each other.

Adaptability has inconsistent definitions in the literature depending on the context. Varying from the ease of change through human modifications to software ability to autonomously adapt to changes (Subramanian et al., 2001). In this paper, we will be focusing on adaptability as an ease that a system can be adapted to the changing business requirements. Modifiability and adaptability are closely interrelated. Adaptability is a non-functional requirement (Chung et al., 2003)

Adaptability is crucial for software businesses in modern rapidly evolving environments. Many businesses seek opportunities and new ways to generate revenue by developing adaptable software systems. The requirements are constantly changing and developing the solution that satisfies the client requires adaptability to succeed. (Fayad and Cline, 1996) Kassab et al. (2018) conducted an empirical study in which participants reported the main challenges in designing architecture. The study concluded that approximately 50% of the participants reported continuous changes in requirements to be the main challenge.

Overall quality attributes should be the guiding sign when designing software. As said before, requirements will change, and designing architecture based on them will certainly lead to dead ends. A study by Kassab et al. (2018) study showed that 62%, out of those who designed architecture based on quality, were satisfied with the software's architectural ability to achieve project goals. In addition, the group that didn't design based on quality was 43% satisfied with the architectural ability of the software. Indicating the importance of quality attributes in designing the architecture.

3.3 Tactics

In this section we will be discussing the architectural tactics and decisions which are known to enhance the wanted quality attributes promoting adaptability. These tactics will help the decision-making in the projects' early phases. However, these tactics should be evaluated based on the previously mentioned risk-driven model. When designing a system based on quality attribute requirements, it is important to consider the parts of the system that will affect this attribute. Bass et al. (2012, p. 117) provide four questions to help when designing for modifiability – “What can change? What is the likelihood of the change? When is the change made and who makes it? What is the cost of the change?”.

Discussing these questions will help to prioritize the elements of the software. This draws the focus to the most important element and its ability to handle changes.

Before going forward, it is necessary to understand basic design concepts. These concepts will help to reason why certain decisions support modifiability and adaptability. Bass (2012) explains that modifiability can be improved by utilizing coupling cohesion and the size of a module. Coupling is a measurable variable that indicates the probability of a modification in one module affecting other modules. High coupling increases the complexity of implementing modifications thus making changes more costly. On the other hand, cohesion simply informs how cohesive two modules are. Higher cohesion means it is less likely that specific change will impact multiple responsibilities. (Bass et al., 2012, p. 121)

Coupling reduction between two modules decreases the costs of modifications made in these modules. Encapsulation is one way to reduce the coupling between modules. This means introducing an application programming interface (API) to a module. Now modules interact through APIs, which serve boundaries and limit the responsibilities. These restrictions reduce coupling because modules don't rely on or interfere with other modules. Another way to decouple modules is to use an intermediary that cuts off dependencies between modules. (Bass et al., 2012, p. 124) Implementing middleware removes the need for one module to understand the insights of another module, instead it's only needed to know the middleware to communicate. One can prevent coupling by restricting dependencies between modules. This can be utilized by limiting the visibility of the module or setting access to only authorized modules. These restrictions limit developers' ability to work only with specific modules. Bass (2012) also lists refactoring modules and abstracting common services as tactics to reduce coupling. Refactoring involves identifying duplicate code in modules and creating new submodules to replace the duplicated sections. However, on a more abstract level, if two modules provide similar services it could be more cost-effective to combine them into one module. This would lead to modifications that affect one service, appearing only in the combined module. As previously stated, changes affecting only one module provide more modifiability to the software (Bass et al., 2012, p. 124).

Modifiability can be enhanced by investigating module sizes. It is obvious that if a certain module is overly large, the modifications to this module take more time and more money (Bass et al., 2012, p. 123). Very large modules with multiple services increase the coupling between them making it beneficial to split the module into separate modules. However, Bass et al. (2012, p. 121) emphasize the need for a clear understanding of the change behind module size reduction.

Increasing cohesion enhances modifiability which can be done by inspecting module responsibilities and possibly removing or moving them from the module. A module that has multiple responsibilities should be the main target of this investigation. For instance, if these responsibilities don't operate for the same intention, they should be moved to another module. Possibly creating a new module for the responsibility could be the solution here. (Bass et al., 2012, p. 123)

Tactics discussed in this section provide solid and high-quality structure for the architecture, which can be cost-effectively refactored when needed. These low-level architecture tactics are used to improve modifiability and adaptability in software. By understanding these tactics, we can better reason why certain architectural patterns enhance different quality attributes.

4. ARCHITECTURE AND DESIGN PATTERNS FOR WEB APPLICATIONS

In this chapter, we will be discussing different high-level architectural patterns such as layered architecture and microservices. We will briefly explain the low-level design pattern Model-View-Controller. Starting from the less modularized architecture to modularized and finally to the physically separated architectural patterns. As Gonçalves et al. (2021) explain the modularization of monolith architecture provides a middle step towards microservices architecture. These patterns are commonly used in web applications.

4.1 MVC and Monolithic

The Model-View-Controller (MVC) architecture was first introduced in the 1970s. The main idea is to separate the architecture into three modules by their functionalities thus including the wanted characteristics for adaptability. The responsibilities are divided as follows, the model is responsible for communicating with the database and the controller handles the business logic. The controller communicates with the model and view. The View provides the user interface and communication with the user. (Pop and Altar, 2014) However, the MVC design pattern is partition-independent, it provides a low-level architecture pattern to the system. This pattern is introduced in a monolith architecture, meaning an application that runs in a single address space. (Leff and Rayfield, 2001)

The MVC pattern is perfect for the web applications development process, as it involves different technologies that can be split into functional modules. User interaction with web applications relies on actions and responses, which is suitable for the MVC design. The view unit state is changed by the users' action, which then triggers the controller and again model. This cycle provides interaction between the user and the system as shown in Figure 2. (Pop and Altar, 2014)

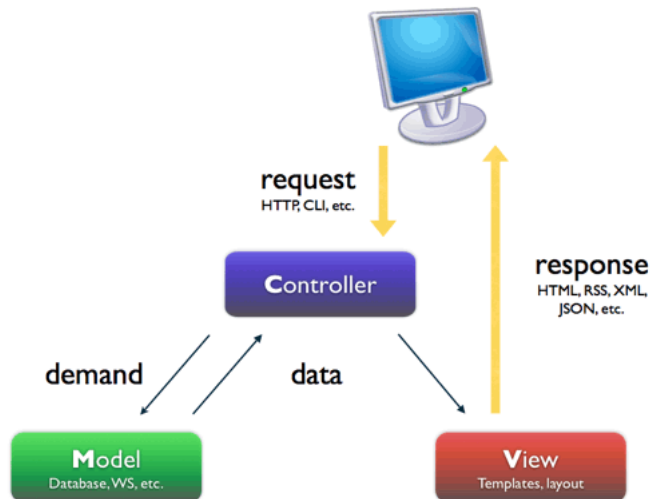


Figure 2: Illustration of the MVC pattern (Stack Overflow, 2011).

The model manages the tasks related to data which includes validation, session state and control, and database structure. This simplifies the controller by removing these responsibilities from it thus making the controller more modifiable. However, it is necessary to keep the model as simple as possible by only including the data processing required for the real-life object. This simplicity provides reusability between applications as the controller includes most of the data processing. The controller handles the events which are triggered by either the user or the system itself. The controller processes the requests and interacts with the model, receiving necessary data and combining it with the view. Overall the MVC architecture is a perfect fit for web applications that operate as a set of HTTP requests and responses. (Pop and Altar, 2014)

Monolith architecture pattern doesn't separate the functional parts into separated deployable services rather they are in one codebase. This has its pros and cons; one is it becomes difficult to operate when the development team grows. However, for rapid prototyping and small teams, this architecture pattern provides simplicity and there isn't much risk involved if low-level modularity is implemented. In small teams, monolith architecture provides adaptability, and developers can implement changes to the system quickly. As previously mentioned, when designing architecture, it is necessary to evaluate the risks and resources involved in the decisions.

Using an evolutionary design style means starting with a monolithic architecture and, as the application and business grow, refactoring core business components into separate services with a defined interface decoupling them. These modules can be managed by different teams. Before implementing a loose-coupled microservices application it is needed to understand the responsibilities of different modules, which is not trivial and

needs refactoring iterations. This is the reason software projects are designed as a monolith because they provide more strongly connected domain entities. Contradictory, not utilizing evolutionary designing will lead to premature modularization and often cause refactoring of these architectural interfaces. This leads to increased costs, as discussed before. (Gonçalves et al., 2021)

4.2 Layered architecture

The layered architecture consists of layers and each layer has a specific role in the application. There can be presentation logic and business logic layers, although there is no specific layer amount. While the layers can be defined to match specific applications, there are still commonly used layers such as presentation, business, persistence, and database layers. Smaller applications might have three layers where business and persistence layers are combined. (Richards, 2015)

The key to layered architecture is to have a strict responsibility and role for each layer. Following this principle, the presentation layer for instance doesn't need information about retrieving customer data as its only responsibility is to display the data in the user interface. Similarly, the business layer doesn't know where the data is coming from, it only requests and receives it from the persistence layer and adds business logic to the data, then sends it to the presentation layer. In figure 3, layered architectural elements are visualized. (Richards, 2015)

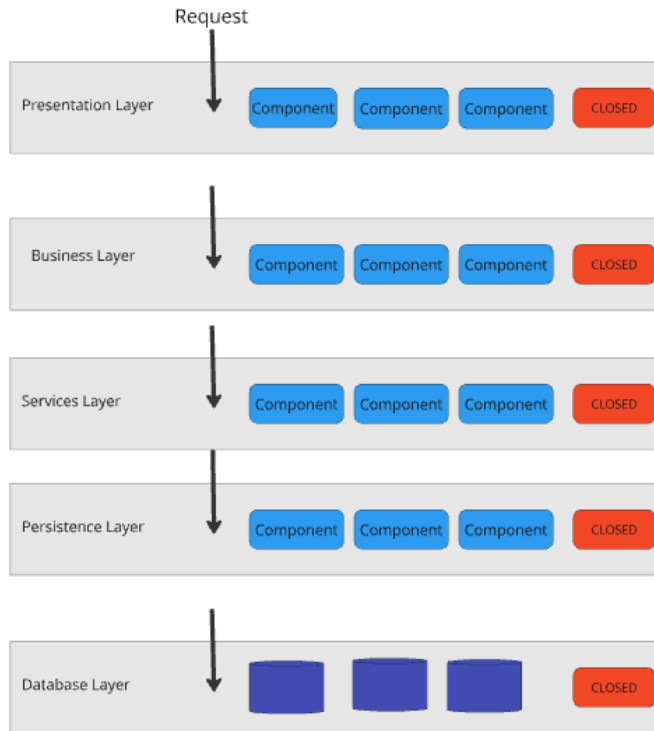


Figure 3: Layered architecture diagram based on the source (Richards, 2015).

These limitations between layers support the adaptability of an application. One of the layer limitations strengthening aspects is the way requests flow through the layers. The layers of isolation concept ensures that changes made in one layer only impact the current layer and possibly its linked layer. However, if the architecture enables the presentation layer access to the persistence layer, the changes made in the persistence layer will now affect not only the business layer but also the presentation layer. Creating tight coupling in the application and interdependencies between components. Which reduces adaptability and modifiability, making it costly to implement changes. (Richards, 2015)

Disadvantages of layered architecture are the situations where requests and response data flow through layers without any logic provided, meaning that they are pass-through. According to Richards, these situations cannot be eliminated. However, if these situations are common in the application, it can be beneficial to open some layers. Opening layers will eventually reduce isolation thus affecting adaptability. (Richards, 2015)

4.3 Microkernel

The key idea of microkernel is to cause separation and isolation between components, making it a beneficial choice for product-based applications. It can be separated into a core system and plug-in components which enables flexibility and extensibility.

Applications that utilize microkernel architecture have separated the minimal viability functionality from the core system containing the general business logic. The plug-in modules contrarily contain additional features that include advanced processing and custom code. (Richards, 2015)

Adaptability in web application software is crucial because often the requirement for new products changes along the way, however, the core functionality can be known. According to Richards (2015), the microkernel architecture provides a core system and as the requirements evolve the system enables the development of new features and functionality without changing the solid core. Requirements often change drastically forcing architectural pattern changes. The microkernel architecture supports flexibility and the ability to refactor, making it suitable for the initial choice of architectural pattern. (Richards, 2015)

4.4 Microservices and Service-oriented Architecture

There are similarities between Service-oriented architecture (SOA) and microservices, meaning that both patterns are based on the idea of physical separation and independence of functional parts of the software. Microservices provide adaptability for the software.

Microservices consist of service components that can be different in size containing one or more modules as shown in Figure 4. These modules are separately deployable units providing adaptability by decoupling them thoroughly. However, this intensive decoupling and freedom of determining the granularity of service components has its pros and cons. Microservices components are fully distributed, and they communicate via communication protocol. (Richards, 2015)

Compared to layered and monolithic non-distributed architectures, microservices enable much better deployment and thus adaptability. As we have discussed the importance of evolutionary design, in this case microservices have evolved from the impracticality of other architectural patterns. Evolution from monolithic layered and service-oriented architecture patterns. Understanding the characteristics that reduces deployability of monolithic applications is crucial, as these traits significantly reduce their adaptability. (Richards, 2015)

Microservices have also evolved from service-oriented architecture. SOA provides many positive aspects that are used also in microservices, but it is more complex, expensive and overkill for most applications, says Richards (2015). SOA is essentially a higher-level version of microservices. This approach partitions the application into a set of

business applications that provide services through communication protocols. SOA architecture is meant for larger enterprises because to handle all business applications, you will need a team of developers per application. However, there are widely conducted studies about SOA, although it provides physical separation of the application at the same time creating an expensive and complex implementation. (Villamizar et al., 2015)

SOA wasn't designed for the modern age of cloud computing, meaning that it is challenging to add or remove servers on demand. At the same time, the ease of implementing changes as new requirements arise becomes harder needing a lot of complex configuration and thus consuming time. (Villamizar et al., 2015) Eventually, as application businesses grow, the increase of developers is inevitable. This requires the separation of monolithic architecture into distributed applications for it to provide quick adaptation to changing requirements.

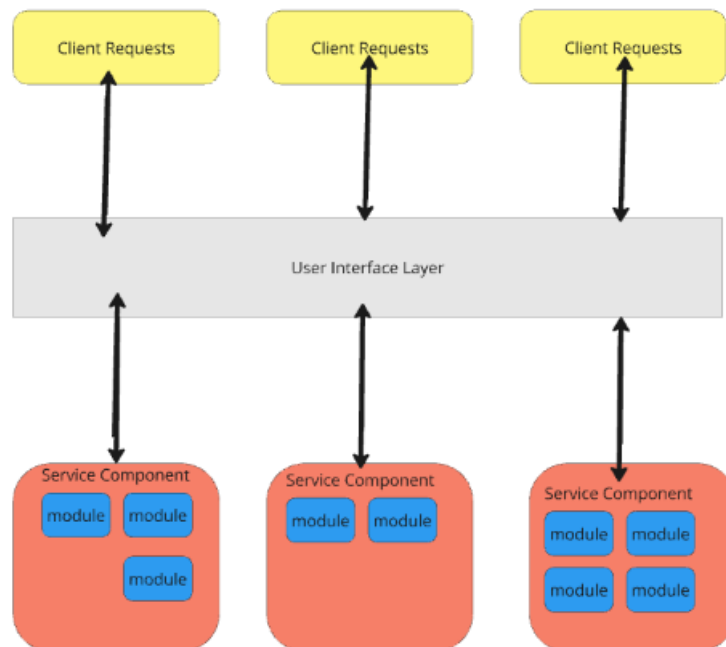


Figure 4: Basic microservice architecture based on the source (Richards, 2015).

The difficulties in microservices design lie in the level of granularity. If service components have too much responsibility and size, the microservice attributes such as deployability and loose coupling will be affected negatively. Conversely designing too many service components, small in size, often needs orchestration which will make the architecture complex and closer to SOA. However, a correctly implemented microservice pattern provides fewer risks in deploying as deployments can be separated providing continuous development. (Richards, 2015)

The benefits of microservices are well-researched in the literature. Bogner et al. (2019) studied the mapping between modifiability tactics and SOA and Microservice patterns, discovering that both patterns contain qualities beneficial for modifiability. They found that approximately 40% of SOA design patterns and 50% of Microservice patterns could be mapped to these modifiability tactics.

Even though microservices provide these benefits and clearly enable adaptability to the architecture, they come with negative effects. We once again conclude that architecture decisions must be based on evaluating risks and the current business state. On paper, microservices provide the perfect solution for adaptability, however it costs to implement and maintain. It requires teams of developers because the service components need complex configurations which take resources. The cloud setup of microservices will increase maintenance costs. (Villamizar et al., 2015)

5. CONCLUSION

Designing web applications for adaptability and business needs is crucial in software development. As discussed in this paper, the requirements determine what the software will eventually provide. However, these requirements tend to vary according to the development project. This is why designing web applications based on quality attributes provides stability to the project. Sources used in this paper state that there is no perfect architecture, and the structure of the software depends on the specific domain but also on the situation that the business is in. Meaning, that resources such as development team size and knowledge but also budget determine where to use time, in designing or implementing.

It is important to understand that every architectural decision needs to be reasoned but also that they require time to design. This is why in some situations the time used in the implementation can be more beneficial. This is a low-risk situation where there is no remarkable risk involved in implementing the software. As said every decision costs, which is why utilizing risk as a guide in decision making is a good way to prioritize limited resources in areas that have potential risks involved.

When prioritizing the ability to prepare for changes, tactics that should be implemented in architecture consist of many aspects, but modularization is important. Modularization introduces aspects to consider, such as Coupling, Cohesion and size of the module. Using tactics to enhance Low-coupling, high cohesion and suitable modules creates a low-level core for the software that promotes its adaptation to changes.

This paper goes from details and root causes to high-level architectural patterns and styles. Understanding the core aspects influencing architecture adaptability helps us reason why certain patterns provide adaptability. All these patterns have positive and negative aspects depending on the scenario. At the beginning of the web application life cycle, when the development team is small and so is the business, the monolith layered architecture is a suitable choice. However, it's important to evaluate certain risks in the implementation, such as the possible changes in the future, which is why the tactics discussed in Section 3.3 play a vital role in the architecture's adaptation capabilities. When low-level tactics are implemented, it is more cost-effective to refactor into micro-services as the business grows.

In conclusion, preparing the software for changing requirements is enabled through low-level partitioning and modularization. However, preparing for everything is time-

consuming and expensive meaning it is not necessary to implement microservices in the early stages of the software life cycle. As it is difficult to truly understand the whole application without implementation and to partition the application into smaller service components, one needs a good understanding of the application. Because the amount and type of service components are the difficult part of designing microservices.

REFERENCES

- Architecture-client-server-model, 2024. URL <https://fitech101.aalto.fi/courses/web-software-development/part-2/2-client-server-model-and-http> (accessed 12.15.24).
- Bass, L., Clements, P., Kazman, R., 2012. *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional.
- Berners-Lee, T., 1996. Hypertext Transfer Protocol -- HTTP/1.0. URL <https://data-tracker.ietf.org/doc/html/rfc1945> (accessed 12.10.24).
- Bogner, J., Wagner, S., Zimmermann, A., 2019. Using architectural modifiability tactics to examine evolution qualities of Service- and Microservice-Based Systems. *SICS Softw.-Intensive Cyber-Phys. Syst.* 34, 141–149. <https://doi.org/10.1007/s00450-019-00402-z>
- Chung, L., Cooper, K., Yi, A., 2003. Developing adaptable software architectures using design patterns: an NFR approach. *Comput. Stand. Interfaces* 25, 253–260. [https://doi.org/10.1016/S0920-5489\(02\)00096-X](https://doi.org/10.1016/S0920-5489(02)00096-X)
- Client-Server overview, 2024. URL https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview (accessed 11.20.24).
- Conallen, J., 1999. Modeling Web application architectures with UML. *Commun. ACM* 42, 63–70. <https://doi.org/10.1145/317665.317677>
- Fairbanks, G., 2010. *Just Enough Software Architecture: A Risk-driven Approach*. Marshall & Brainerd.
- Fayad, M., Cline, M.P., 1996. Aspects of software adaptability. *Commun. ACM* 39, 58–59. <https://doi.org/10.1145/236156.236170>
- Gonçalves, N., Faustino, D., Silva, A.R., Portela, M., 2021. Monolith Modularization Towards Microservices: Refactoring and Performance Trade-offs, in: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. Presented at the 2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C), pp. 1–8. <https://doi.org/10.1109/ICSA-C52384.2021.00015>
- Hassan, A.E., Holt, R.C., 2002. *Architecture Recovery of Web Applications*. ICSE02 May 19-25 Orlando Fla. USA.
- Kassab, M., Mazzara, M., Lee, J., Succi, G., 2018. Software architectural patterns in practice: an empirical study. *Innov. Syst. Softw. Eng.* 14, 263–271. <https://doi.org/10.1007/s11334-018-0319-4>
- Leff, A., Rayfield, J.T., 2001. Web-application development using the Model/View/Controller design pattern, in: *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*. Presented at the Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference, pp. 118–127. <https://doi.org/10.1109/EDOC.2001.950428>
- Pop, D.-P., Altar, A., 2014. Designing an MVC Model for Rapid Web Application Development. *Procedia Eng.* 69, 1172–1179. <https://doi.org/10.1016/j.proeng.2014.03.106>
- Richards, M., 2015. *Software Architecture Patterns*. O'Reilly Media.
- Stack Overflow, 2011. URL <https://stackoverflow.com/questions/5966905/what-is-the-right-mvc-diagram-for-a-web-application> (accessed 12.12.24).
- Subramanian, N., TechnologyDivision, A., Chung, L., 2001. Software Architecture Adaptability: An NFR Approach. *IWPSE 01 Proc. 4th Int. Workshop Princ. Softw. Evol.* 52–61.
- Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S., 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, in: *2015 10th Computing Colombian*

Conference (10CCC). Presented at the 2015 10th Computing Colombian Conference (10CCC), pp. 583–590. <https://doi.org/10.1109/ColumbianCC.2015.7333476>