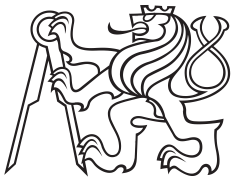
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

# Microkernel architecture and its applicability in application development

**Mikita Aljaksandravič Citarovič**

**Supervisor:** Ing. Kyrylo Bulat
**Field of study:** Software Engineering and Technology
**Subfield:** Enterprise Systems
**May 2024**

# BACHELOR‘S THESIS ASSIGNMENT

## I. Personal and study details

Student's name: **Citarovič Mikita Aljaksandravič** Personal ID number: **511130**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Software Engineering and Technology**

## II. Bachelor’s thesis details

Bachelor’s thesis title in English:

**Microkernel architecture and its applicability in application development**

Bachelor’s thesis title in Czech:

**Architektura mikrojádra a její použitelnost ve vývoji aplikací**

Guidelines:

This bachelor thesis focuses on analyzing microkernel architecture and its applicability in application development. As part of this work, the following tasks will be performed:
- Compare microkernel architecture with other software architecture styles.
- Study successful examples of its application in modern applications.
- Conduct a detailed analysis of the principles, concepts, and components of microkernel architecture.
- Analyze technologies for implementation, design the architecture, and implement the proof of concept application.
- Design test cases and conduct testing of the implemented application.

The bachelor thesis will result in a valuable understanding of the practicality and effectiveness of microkernel architecture in application development. Creating and implementing a proof of concept application plays a crucial role in facilitating the assessment of results from the theoretical analysis. This proof of concept application will act as a tangible demonstration of the explored theoretical principles, allowing for a comprehensive evaluation of their practical implications and potential advantages in real-world scenarios.

Bibliography / sources:

Neal Ford, Mark Richards: "Fundamentals of Software Architecture: An Engineering Approach"
Mark Richards, Neal Ford, Pramod Sadalage, Zhamak Dehghani: "Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures"
Chatley, Robert. (2005). Predictable Dynamic Plugin Architectures.
Chatley, R., Eisenbach, S., Kramer, J., Magee, J., Uchitel, S. (2004). Predictable Dynamic Plugin Systems. In: Wermelinger, M., Margaria-Steffen, T. (eds) Fundamental Approaches to Software Engineering. FASE 2004. Lecture Notes in Computer Science, vol 2984. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-24721-0_9

Name and workplace of bachelor's thesis supervisor:

**Ing. Kyrylo Bulat System Testing IntelLigent Lab FEE**

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **15.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

| Ing. Kyrylo Bulat<br>Supervisor's signature | Head of department's signature | prof. Mgr. Petr Páta, Ph.D.<br>Dean's signature |
|---|---|---|

## III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

| . | |
|---|---|
| Date of assignment receipt | Student's signature |

# Acknowledgements

I would like to express my gratitude to CTU in Prague for the opportunities in education, excellent and supportive teachers, as well as university staff, thanks to whom I was able to gain invaluable experience and knowledge that helped me in writing this work. I would also like to extend my immense gratitude to my supervisor, Ing. Kyrylo Bulat, for his assistance and support during the writing of this thesis; without him, this work would not have been possible. Additionally, I express my sincere thanks to friends, family, and colleagues for their support and understanding during the writing of this thesis.


# Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 22, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 22. května 2024

# Abstract

Since the establishment of software engineering as a professional discipline, there has been a consistent demand for standardization of software development methodologies. Software architecture, encompassing the organization and hierarchy of system elements, has always been a pivotal concern in this field. Historically, early software systems predominantly followed what is now recognized as the monolithic approach. However, by the 1970s, doubts began to surface regarding the universality of this monolithic paradigm, leading to a shift towards the development of more modular systems.

The term "microkernel" was coined in 1981, gaining traction throughout the 1980s. Since then the popularity of this approach has waned, resulting in a decrease in published works and research in the area. However, many applications today still inadvertently employ aspects of microkernel architecture, and solutions to similar problems often bear striking resemblances to those devised in the 1980s.

The purpose of this study was to conduct an analysis of contemporary approaches to microkernel architecture. This work contains a comprehensive examination of fundamental principles, concepts, and components of microkernel architecture, supplemented by illustrative examples. As a proof of concept, a prototype application utilizing microkernel architecture has been developed. The process of prototype creation confirmed several advantages associated with this approach, notably its high modularity, extensibility, and fault tolerance.



**Keywords:** microkernel architecture, plugin architecture, software architecture, webhooks, spring boot, java

**Supervisor:** Ing. Kyrylo Bulat



# Abstrakt

Od zavedení softwarového inženýrství jako profesní disciplíny existuje konzistentní poptávka po standardizaci metodik vývoje softwaru. Softwarová architektura, zahrnující organizaci a hierarchii prvků systému, byla vždy klíčovým zájmem v této oblasti. Historicky se rané softwarové systémy převážně řídily tím, co je nyní uznáváno jako monolitický přístup. Nicméně již v 70. letech začaly vznikat pochybnosti o univerzálnosti tohoto monolitického paradigmatu, což vedlo k přechodu k vývoji modulárnějších systémů.

Termín "mikrojádro" byl poprvé použit v roce 1981 a získal na popularitě během 80. let. Od té doby však popularita tohoto přístupu upadla, což vedlo k poklesu publikovaných prací a výzkumu v této oblasti. Nicméně mnoho aplikací dnes nepřímo využívá prvky architektury mikrojádra a řešení podobných problémů často připomínají ty, které byly vyvinuty v 80. letech.

Cílem tohoto studia bylo provést analýzu současných přístupů k architektuře mikrojádra. Tato práce obsahuje komplexní zkoumání základních principů, konceptů a komponent mikrojádrové architektury, doplněné ilustrativními příklady. Jako důkaz konceptu byla vyvinuta prototypová aplikace využívající architekturu mikrojádra. Proces tvorby prototypu potvrdil několik výhod spojených s tímto přístupem, zejména jeho vysokou modularitou, rozšiřitelností a odolností vůči chybám.



**Klíčová slova:** architektura mikrojádra, architektura pluginů, softwarová architektura, webhooks, spring boot, java

**Překlad názvu:** Architektura mikrojádra a její použitelnost ve vývoji aplikací

# Contents

# Figures



# Tables

# Listings

May 22, 2024

# Chapter 1

# Introduction

The quest to find and standardize best practices and approaches has always been an integral part of the software development industry. One of the objects of such activity is software architecture. The first discussions on the topic emerged in the late 1960s. The term "software architecture" itself was first coined at a NATO-sponsored conference on software engineering techniques in 1969. Notable figures among the conference attendees included Edsger W. Dijkstra and Niklaus Wirth. At that time, the term "architecture" was used in the context of the physical structure of computer systems, i.e hardware architecture. By 1990s the concept of software architecture as a separate discipline emerged[1].

At the inception of modern computing, software architectures predominantly followed a naive monolithic design philosophy. Monolithic architectures encapsulated the entire functionality of an application within a single codebase, often resulting in tightly coupled systems that were challenging to extend, scale, or maintain. However, as the software landscape matured and the demands on applications became more diverse and dynamic, the limitations of monolithic architectures became increasingly apparent.

Discussion of alternative approaches capable of addressing the problems associated with monolithic architecture began as early as the 1970s[2]. Among the emerging approaches, microkernel architecture stands out. The term "microkernel" specifically appeared no later than 1981 when it was used by Richard F. Rashid[3]. Microkernel architecture reached its peak popularity in the late 1980s and early 1990s before declining. In our opinion, this decline was somewhat unjustified, and within the scope of this work, we attempted to revive interest in this approach and demonstrate its applicability in the modern world.

## 1.1 Motivation

The selection of appropriate software architecture is a crucial decision in the development process, as it significantly affects the scalability, maintainability, and overall performance of the system. Among various architectural approaches, the microkernel architecture is a compelling option due to its modular approach. This work aims to explore the details of microkernel

architecture and explain its practical implications in software development.

The research aims to address the increasing need for software systems that are robust, adaptable, and extensible as technology advances. Developers face challenges in creating software that can seamlessly adapt to changing requirements and environments. The microkernel architecture, which isolates core functionalities into independent modules, offers a promising solution to these challenges. The aim is to gain a thorough understanding of the strengths and potential drawbacks of this approach by analysing its architecture and real-world applications in an objective manner.

## 1.2 Objectives

The primary objectives of this work are as follows:

1. **Compare with other architectures.**

   Conduct an evaluative and conceptual comparison of microkernel architecture with monolithic, service-oriented, and microservices architectures. This comparative analysis, in conjunction with the other objectives, will help more accurately identify the strengths and weaknesses of microkernel architecture, as well as its applicability in application development.

2. **Examine case studies and industry applications.**

   Examine existing case studies and real-world applications where microkernel architecture has been implemented successfully. This will enable a better understanding of the strengths of microkernel architecture in real working applications.

3. **Analyse microkernel architecture.**

   To provide an in-depth examination of the microkernel architecture, exploring its core principles, concepts and components. This detailed analysis aims to provide a thorough understanding of how microkernel architecture functions at its foundational level, shedding light on its unique characteristics and design principles.

4. **Practical application in software development.**

   Utilising the acquired knowledge, develop a proof-of-concept prototype of an application with a microkernel architecture. This prototype will showcase the practical application of microkernel architecture and its advantages in specific scenarios.

## 1.3 Architectures Overview

This section provides a brief introduction to monolithic, microkernel, service-oriented and microservices architectures for further comparison.

### 1.3.1 Monolithic Architecture

Monolithic architecture is a classic approach. All application functionality is integrated into a seamless unit. This means that all components such as the user interface, business logic and data access are combined into a single executable or code base.[4, 5]

**Figure 1.1** illustrates the basic topology of a monolithic layered architecture.

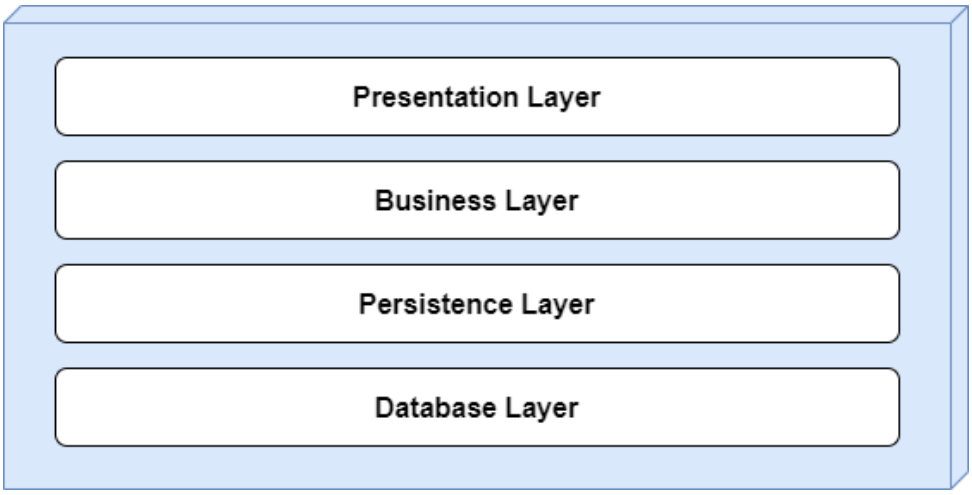



**Figure 1.1:** Basic topology of monolithic layered architecture.[6]

### 1.3.2 Microkernel Architecture

In a microkernel architecture, the system core implements a minimum set of basic functions, and additional functionality is attached in the form of plug-ins. This provides a high degree of flexibility, as developers can extend or modify the system's functionality by adding or removing plug-ins without having to modify the core.[6]

**Figure 1.2** illustrates the basic topology of a microkernel architecture.

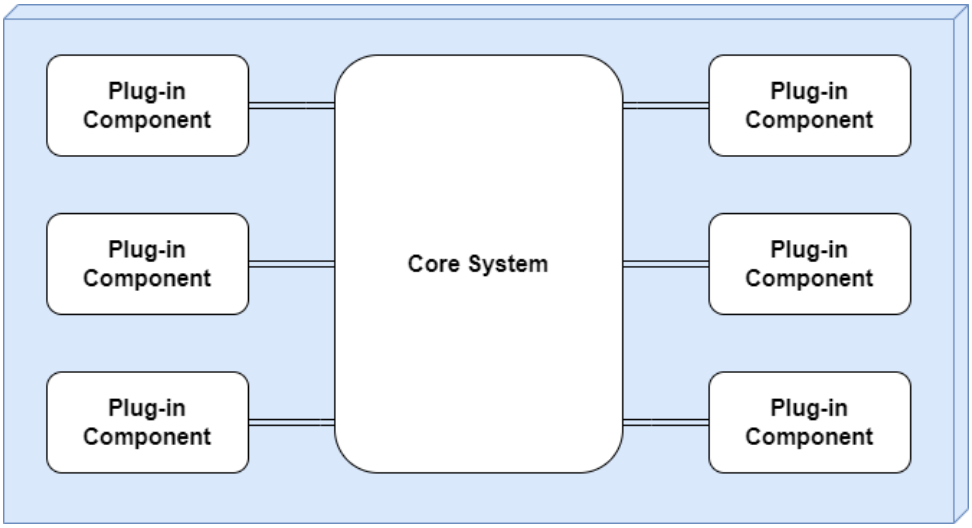



**Figure 1.2:** Basic topology of microkernel architecture.[6]

### 1.3.3 Service-Oriented Architecture

In a service-oriented architecture, an application is divided into independent services that provide specific functionality. Services interact with each other through explicit interfaces, and a central component (service bus) facilitates communication between them. This allows better isolation of functionality and increases system flexibility.[6]

**Figure 1.3** illustrates the basic topology of a service-oriented architecture.

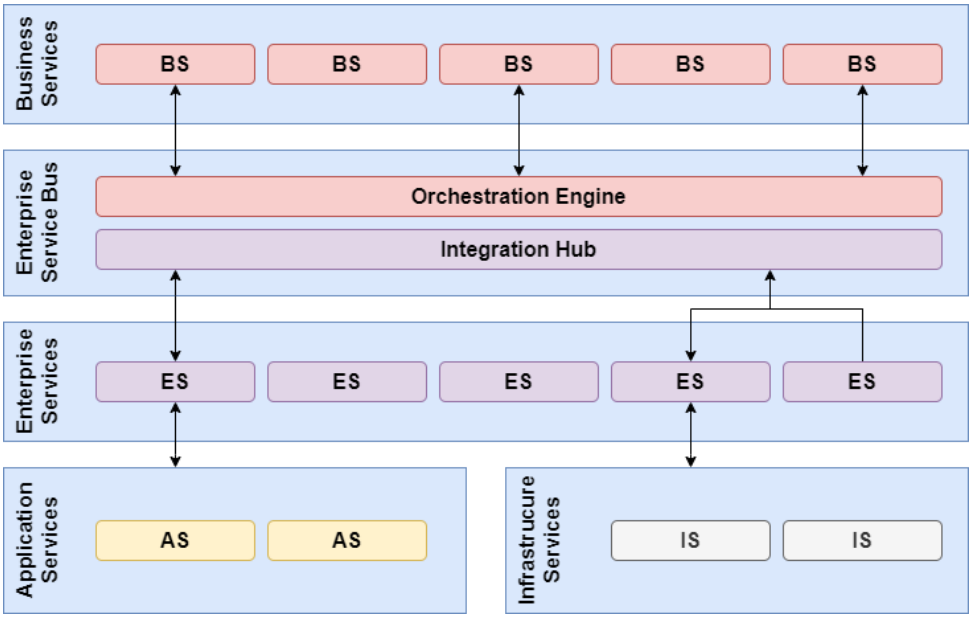



**Figure 1.3:** Basic topology of service-oriented architecture.[6]

### 1.3.4 Microservices Architecture

A microservices architecture breaks down an application into small, self-contained services, each responsible for a specific domain. Services can be developed, deployed and scaled independently. This simplifies upgrades and maintenance, and allows different technologies to be used for each service.[5]

**Figure 1.4** illustrates the basic topology of a microservices architecture.

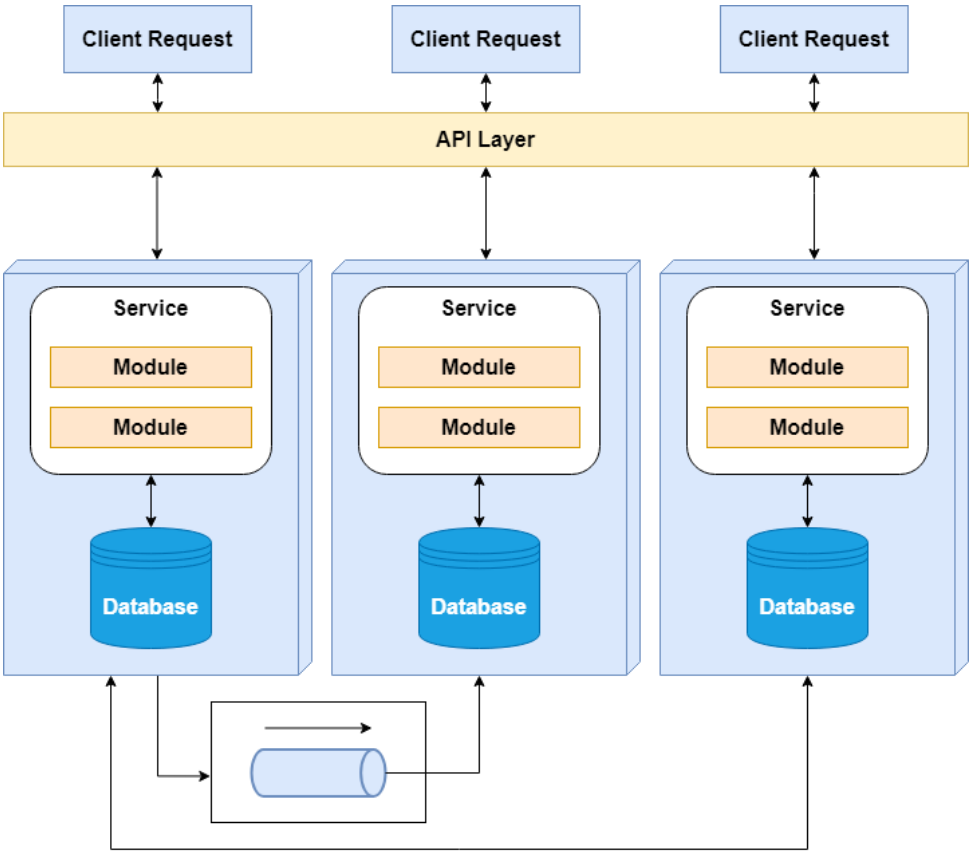



**Figure 1.4:** Basic topology of microservices architecture.[6]

# Chapter 2

# Analysis

In this chapter, we will conduct an analysis of microkernel architecture, comparing it with other architectural approaches based on specific criteria and their conceptual differences. We will also examine successful use cases of applying this architecture in the development of modern applications. In conclusion, a more detailed analysis of microkernel architecture will be carried out from the perspective of principles, concepts, and its components. This analysis will provide a deeper understanding of the principles of microkernel architecture, revealing its unique characteristics and design principles, which will be useful to us in developing a prototype application later on.

## 2.1 Comparison and Evaluation

In this section, we will conduct a comparative analysis of microkernel architecture with other popular architectural approaches, which will bring us closer to a better understanding of the features and advantages of microkernel architecture in various software development scenarios.

### 2.1.1 Comparative Evaluation Criteria

To assess the microkernel architecture and compare it with other architectural approaches, it is necessary first to define the criteria for the comparative evaluation of these architectural approaches.

The following non-functional requirements have been chosen as evaluation criteria [1]:

1. **Deployability.**

   **Definition:** The ease with which the architecture can be deployed in different environments.

   **Importance:** A highly deployable architecture ensures flexibility and adaptability across various systems.

2. **Elasticity.**

[1] these criteria were taken from the book [6]

**Definition:** The ability of the architecture to dynamically scale resources based on demand.

**Importance:** Elasticity is crucial for handling varying workloads efficiently and optimising resource utilisation.

3. **Evolutionary.**

   **Definition:** The capacity of the architecture to evolve and adapt to changing requirements over time.

   **Importance:** An evolutionary architecture supports long-term sustainability and accommodates future enhancements.

4. **Fault Tolerance.**

   **Definition:** The system's ability to continue operating and providing services in the presence of faults or errors.

   **Importance:** Fault tolerance ensures system reliability and minimises disruptions due to unexpected failures.

5. **Modularity.**

   **Definition:** The extent to which the architecture is divided into independent and interchangeable modules.

   **Importance:** Modularity enhances maintainability, as changes in one module do not affect others, promoting code reusability.

6. **Overall Cost.**

   **Definition:** The total cost associated with developing, deploying, and maintaining the architecture.

   **Importance:** A consideration of the overall cost helps in making informed decisions regarding resource allocation and budgeting.

7. **Performance.**

   **Definition:** The efficiency and speed with which the architecture executes its functions.

   **Importance:** Performance is critical for achieving responsiveness and meeting user expectations.

8. **Reliability.**

   **Definition:** The ability of the architecture to consistently perform its intended functions without failures.

   **Importance:** Reliable systems build trust among users and stakeholders, ensuring consistent and dependable operation.

9. **Scalability.**

   **Definition:** The capability of the architecture to handle an increasing workload by adding resources.

**Importance:** Scalability is vital for accommodating growth and preventing performance degradation under higher loads.

10. **Simplicity.**

    **Definition:** The degree of simplicity and clarity in the design and implementation of the architecture.

    **Importance:** A simple architecture is easier to understand, maintain, and troubleshoot, reducing the likelihood of errors.

11. **Testability.**

    **Definition:** The ease with which the architecture can be tested to ensure its correctness and reliability.

    **Importance:** Testability facilitates the identification and resolution of issues during development and maintenance phases.

When comparing architectural approaches for each criterion, a rating will be assigned on a scale of low, medium, to high, where "low" indicates that the architecture does not meet the characteristic well, "medium" indicates moderate alignment, and "high" indicates that it is one of its primary advantages.

### 2.1.2 Comparative Evaluation

Based on the selected evaluation criteria, let's conduct a comparative analysis of microkernel architecture with monolithic, service-oriented, and microservices architectures.

#### Monolithic Architecture

1. **Deployability** (low)

   Monolithic architectures, although easier to deploy in a single environment with less configuration, often present challenges during deployment. The deployment process for a monolith can be complex, requiring the entire application to be deployed as a single unit. This can result in increased risks and efforts in preparing for deployment[6, 5, 7, 4].

2. **Elasticity** (low)

   Monolithic architectures have limitations in scaling, especially in terms of horizontal scaling. Scaling the entire monolith can be challenging because resources are allocated to the entire application rather than specific components[5, 7].

3. **Evolutionary** (low)

   Due to their tightly integrated nature, monolithic architectures can be difficult to adapt to changing requirements. Changes and updates may require changes to the entire application. This makes it less flexible in responding to changing business needs[7, 4].

4. **Fault Tolerance** (low)

   Due to its monolithic nature, a monolithic architecture has low fault tolerance. A failure in one part of the system can easily break the entire system[4, 7].

5. **Modularity** (low)

   Although it is possible to structure a monolithic application with internal modularity, it is usually deployed and maintained as a single, tightly coupled unit. This reduces the ability to independently update or replace specific modules[4].

6. **Overall Cost** (high)

   The monolithic architecture is simple in terms of development, deployment, and maintenance. However, it is important to note that as the application grows, the overall complexity and resulting cost increase significantly[4].

7. **Performance** (high)

   Monolithic architectures often have a high level of performance because all of the elements of the system are tightly integrated into a single unit. As a result, communication between components is usually more efficient than in distributed systems[4].

8. **Reliability** (medium)

   Monolithic architectures are typically reliable, but there are risks associated with the deployment and testing process. As a result, rigorous testing is essential to ensure the reliability of the entire application[7].

9. **Scalability** (low)

   In monolithic architectures, scalability is typically limited to vertical scaling, which involves adding more resources (CPU, memory) to a single node to scale the entire application. Horizontal scaling, which distributes the load across multiple nodes, is more challenging[7, 5].

10. **Simplicity** (high)

    Monolithic architectures are relatively simple to develop and maintain. However, as the application grows and becomes more complex, this simplicity can diminish[4, 7].

11. **Testability** (medium)

    Although testing individual components may seem easier, testing the entire application as a single unit can be challenging[4, 6].

### Microkernel Architecture

1. **Deployability** (medium)

   The modular nature of microkernel architecture enables a high level of deployability, where the system core and plugins can be developed, tested, and deployed independently of each other. However, deployment of the core system can be difficult and risky due to its monolithic nature[6].

2. **Elasticity** (low)

   As for flexibility, due to the architectural characteristics of microkernel architecture, where the core is a critical component of the system, this criterion is not well-supported[6].

3. **Evolutionary** (high)

   Microkernel architectures are designed to be modular, allowing for easier evolution and adaptation to changing requirements. This modularity enables the development of new plugins or components without necessarily affecting the entire system[8].

4. **Fault Tolerance** (medium)

   Microkernel architectures enhance fault tolerance through component isolation. If a plugin or component fails, it should not affect the entire system. However, if the system core fails, the entire application may cease to function[6].

5. **Modularity** (high)

   Modularity is a fundamental characteristic of microkernel architectures. The system is designed to have a minimal core and additional plugins or modules that can be added or modified independently[8].

6. **Overall Cost** (medium)

   In the process of development it is necessary to think over the design of the system in advance, because in case of any errors in terms of design their correction in the future can lead to big problems, especially when the application is already working and modules are written for it by third-party developers. Backward compatibility issues will also have to be addressed during system support[6].

7. **Performance** (medium)

   Microkernel architectures may introduce some overhead due to inter-component communication[6].

8. **Reliability** (medium)

   Modularity in microkernel architectures enhances reliability by isolating components, preventing failures in one part of the system from affecting others. However, a level of dependency is introduced by relying on a central core[6, 8, 9].

9. **Scalability** (low)

   Like monolithic architectures, microkernel architectures may have limitations in scalability due to the central nature of the core, which can pose challenges in scaling the system horizontally[6].

10. **Simplicity** (high)

    The microkernel architecture emphasises simplicity by focusing on essential services in the core. The modular structure contributes to a clear and understandable overall design, further enhancing the simplicity of the microkernel architecture[6, 9].

11. **Testability** (high)

    The modularity of microkernel architectures enhances testability. Each module, including the core, can be tested independently, making it easier to identify and fix problems in specific components without affecting the entire system[6, 9].

### Service-Oriented Architecture

1. **Deployability** (low)

   Deploying Service-Oriented Architecture (SOA) can be challenging. Developing and deploying services independently can add complexity and coordination overhead[10].

2. **Elasticity** (medium)

   The capacity to independently scale services enhances adaptability, but coordinating and managing these autonomous components can restrict its full potential[10].

3. **Evolutionary** (low)

   The modular nature of services enables updates, but maintaining compatibility and orchestrating changes across services can be challenging[10].

4. **Fault Tolerance** (medium)

   The use of independent services can mitigate the impact of failures. However, vulnerabilities may still arise due to coordination and dependencies between services[10].

5. **Modularity** (medium)

   The challenges of coordinating and managing the interactions between services affect the overall modularity, even though the services are designed as independent modules[10].

6. **Overall Cost** (low)

   Due to the distributed nature and complex topology of SOA, the overall cost of development, deployment, and maintenance is high[10].

7. **Performance** (low)

   SOA may have limitations in terms of performance due to the distributed nature of services and associated communication overhead, which can result in lower performance[6].

8. **Reliability** (low)

   Although the independence of services can enhance localised reliability, challenges in coordination and potential points of failure across services can negatively impact the overall system reliability[10].

9. **Scalability** (high)

   The capacity to scale services independently according to specific requirements contributes to scalability, despite potential coordination challenges[10].

10. **Simplicity** (low)

    SOA is a distributed architecture with a complex topology, which makes it challenging to develop, deploy, and maintain. The management of communication between services, handling dependencies, and coordinating updates all contribute to its low simplicity[6].

11. **Testability** (low)

    Coordination challenges and the need for comprehensive integration testing may hinder the independent testing of services, which can impact the overall testability of the architecture[6].

### Microservices Architecture

1. **Deployability** (high)

   Microservices architecture is known for its deployability. The development and deployment of independent microservices allow for a high degree of flexibility and agility. Contributing to a seamless deployment process, changes to individual microservices can be deployed without affecting the entire system[5].

2. **Elasticity** (high)

   Each microservice can be scaled independently based on specific demands, enabling efficient resource allocation and adaptability to varying workloads. This scalability enhances the overall responsiveness of the architecture[7].

3. **Evolutionary** (high)

   The modular nature of microservices makes updating and introducing new services easy without disrupting the entire system. This flexibility supports continuous evolution of the architecture in response to changes in business requirements[5].

4. **Fault Tolerance** (high)

   The use of microservices reduces the impact of failures, as issues in one microservice do not affect others. This decentralisation contributes to a robust and fault-tolerant system[4].

5. **Modularity** (high)

   Each microservice is designed as an independent module with well-defined interfaces, promoting flexibility and ease of management through a high level of modularity. This approach allows for independent development, maintenance, and updates[5].

6. **Overall Cost** (low)

   The distributed nature of microservice architecture makes it difficult to properly divide into bounded contexts and deal with data consistency issues, resulting in high overall costs[5].

7. **Performance** (low)

   Microservices architecture can introduce performance overhead due to inter-service communication[5]. However, this impact is often outweighed by the benefits of scalability and fault tolerance. Efficient design and communication strategies can mitigate performance concerns.

8. **Reliability** (high)

   The reliability of a system is enhanced through the isolation of services in a Microservices architecture. This is because failures in one microservice do not affect others, resulting in a highly reliable system. Additionally, the decentralised nature of microservices minimises the risk of system-wide outages[7, 5, 4].

9. **Scalability** (high)

   The architecture's ability to independently scale each microservice is a key strength, enabling efficient resource utilisation and responsiveness to varying workloads[7, 4, 5].

10. **Simplicity** (low)

    Microservices architecture can introduce complexities in managing distributed systems, inter-service communication, and potential consistency challenges. Although powerful and flexible, coordinating and orchestrating microservices can make it less simple compared to monolithic or less distributed architectures[5].

11. **Testability** (medium)

    Each microservice can be tested independently, enabling effective unit testing and isolation of potential issues. This modular approach simplifies the identification and resolution of problems during the testing phase,

contributing to the overall reliability of the system. Nevertheless, considering that microservice architecture is distributed, testing the system as a whole can be quite a challenging task and requires automation[5, 4].

## Overall Comparison

**Table 2.1** shows the final comparison of the architectures based on the specified criteria.

**Table 2.1:** Overall comparison of architectural approaches.

| Criteria | Monolith | Microkernel | SOA | Microservices |
|---|---|---|---|---|
| **Deployability** | low | medium | low | high |
| **Elasticity** | low | low | medium | high |
| **Evolutionary** | low | high | low | high |
| **Fault Tolerance** | low | medium | medium | high |
| **Modularity** | low | high | medium | high |
| **Overall Cost** | high | medium | low | low |
| **Performance** | high | medium | low | low |
| **Reliability** | medium | medium | low | high |
| **Scalability** | low | low | high | high |
| **Simplicity** | high | medium | low | low |
| **Testability** | medium | high | low | medium |

When comparing a monolithic architecture with a microkernel architecture, it is evident that the main difference lies in the fact that the core in a microkernel architecture typically implements only the minimum necessary functionality. This allows for easy extension or modification through plug-ins, resulting in a high level of modularity and evolutionary capability. However, this comes at the cost of performance and simplicity.

Comparing microkernel architecture with service-oriented and microservices architectures may seem strange, as the former is of the monolithic type and the latter are of the distributed type. However, microkernel architecture can also be distributed in the context of remote plugin connectivity.

One can find some similarities between microkernel architecture and service-oriented architecture. For instance, in service-oriented architecture, there is a central component responsible for communication between services called a service bus. In contrast, in microkernel architecture, the core is responsible for this. Both architectures discourage direct communication between services/plugins.

When comparing microkernel and microservices architectures, it is evident that in the context of microservices architecture, there is no central node, communication between services usually takes place directly or through a message broker.

It is important to note that among the compared architectures, only the microkernel architecture provides mechanisms at the conceptual level for extending or modifying core behaviour. This feature makes the microkernel architecture appealing in situations where a flexible system that can adapt to different usage scenarios is required.

## 2.2 Case Studies and Industry Applications

Microkernel architecture has found practical application in a variety of domains, demonstrating its effectiveness in diverse contexts. Below are notable case studies and industry applications showcasing the versatility and benefits of microkernel-based systems.

### 2.2.1 Eclipse IDE

Eclipse, a widely used open-source IDE, uses a microkernel-based architecture known as the Rich Client Platform (RCP). The RCP separates the core functionality (the microkernel) from the rest of the Integrated Development Environment (IDE), allowing for modularity and extensibility through plug-ins. This design choice allows developers to customise their development environment and seamlessly integrate additional functionality[11].

### 2.2.2 GNU Emacs

GNU Emacs is a text editor that is highly extensible and versatile. Its microkernel-like architecture sets it apart from other text editing tools. The core functionality provides essential text editing capabilities, while Emacs's strength lies in its extensibility through Emacs Lisp. This extensible design allows users to tailor Emacs to their specific needs, seamlessly integrating additional features and functionalities[12].

### 2.2.3 Shopify

Shopify is a leading e-commerce platform known for its microkernel-inspired architecture, which has been instrumental in its success. The microkernel at the core of Shopify handles essential e-commerce functionalities, providing a strong foundation for online businesses. What distinguishes Shopify is its extensive ecosystem of plugins and apps that enable users to customise and extend their e-commerce stores.

Shopify's architecture is unique in that it is built around the capability for third-party developers to create plugins[13]. This approach has turned the platform into a thriving marketplace for a diverse range of applications, from payment gateways to marketing tools. The microkernel architecture of Shopify is not only modular but also distributed. Third-party plugins are developed and maintained on external servers and interact with the core system through well-defined Application Programming Interfaces (APIs).

This distributed microkernel architecture provides several advantages to Shopify. It ensures that the core remains lightweight and efficient as additional functionalities are handled externally. Additionally, it allows for seamless updates and improvements to the platform without disrupting the services provided by third-party plugins. The platform's versatility has been enhanced by the thriving ecosystem of developers contributing to Shopify's plugin

marketplace, which has fueled its continuous evolution as a leading e-commerce solution. The success of Shopify is closely linked to its microkernel-inspired architecture, which allows businesses to customise their online stores with a wide range of external plugins.

**Figure 2.1** illustrates the relationship among apps, merchants, developers, and Shopify.

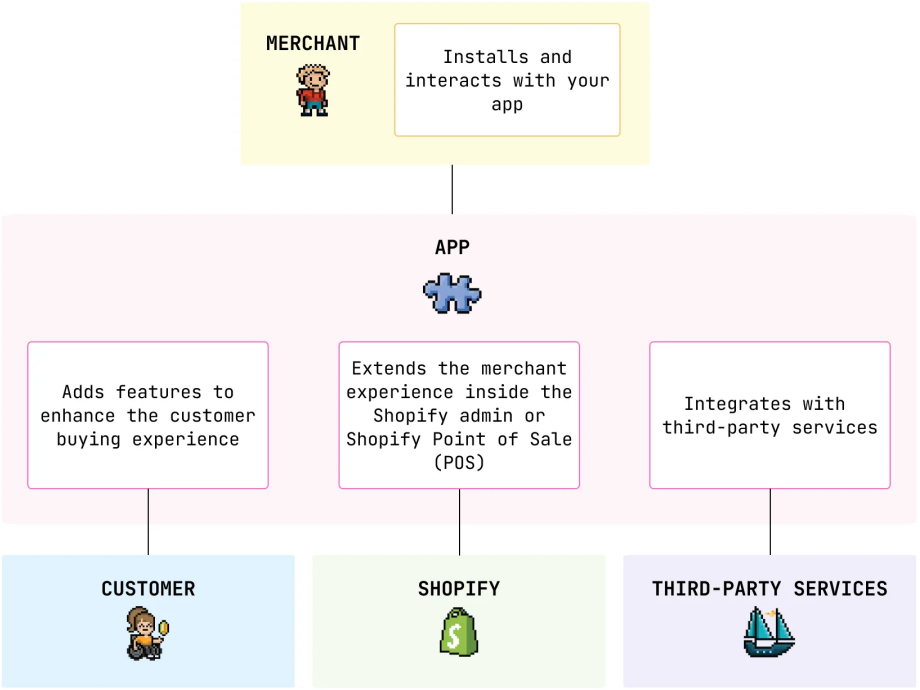



**Figure 2.1:** Relationships between apps, merchants, developers, and Shopify[13].

**Figure 2.2** shows the impact of the app on the customer experience, the user experience of the shop owner in the Shopify admin panel, and the integration with third party services.

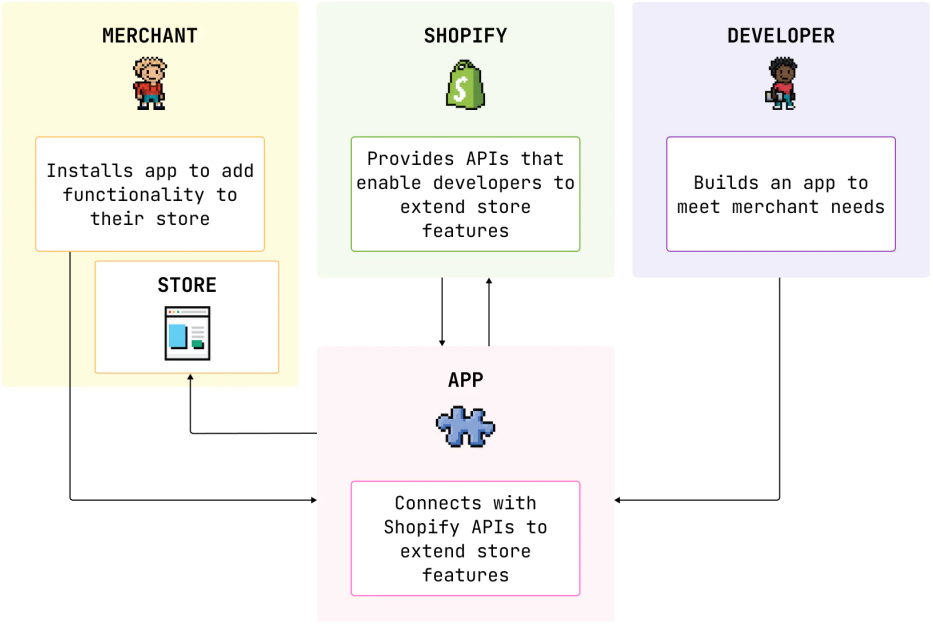



**Figure 2.2:** Integrating the app within Shopify[13].

### 2.2.4 Wordpress

WordPress is a popular Content Management System (CMS) with a microkernel-like architecture that has contributed to its widespread use. The core engine of WordPress provides fundamental content management capabilities. What distinguishes WordPress is its extensive ecosystem of plugins, which allows for robust extensibility. The modular approach of WordPress allows users to easily extend and customise their websites by integrating additional features, such as e-commerce solutions and Search Engine Optimization (SEO) tools[14, 15].

The ability to augment the core functionality with plugins has been a significant factor in WordPress's popularity among users and developers alike. The extensibility of WordPress has democratised website creation, allowing users with varying technical expertise to customise their sites to specific needs without the need for complex coding. The thriving community of plugin developers has further enriched the WordPress ecosystem, fostering innovation and continually expanding the platform's capabilities. The enduring success of WordPress can be attributed to its microkernel-inspired architecture and extensibility, which have made it the preferred CMS for millions of websites worldwide.

### 2.2.5 Chromium

Chromium is the open source project behind the widely used Google Chrome browser. It features a microkernel-inspired multi-process architecture. Core functionality is divided into multiple processes, each responsible for specific tasks, increasing both security and stability. In particular, Chromium's extensibility is a key strength, allowing developers to extend its capabilities through a rich ecosystem of plug-ins. This microkernel approach enables seamless integration of additional features and functionality, empowering users to customise their browsing experience and fostering a thriving community of extensions that contribute to the browser's versatility and user experience[16, 17].

### 2.2.6 Yarn

Yarn is a widely used JavaScript package manager that follows a microkernel architecture. Its lightweight core focuses on essential package management tasks. What sets Yarn apart is its emphasis on extensibility through a robust plugin system. This allows developers to customise their package management workflows with a variety of plugins, ensuring flexibility and efficiency[18, 19].

It's interesting to note that plugin support only appeared in the second version of Yarn. Here is what Maël Nison, one of the Yarn maintainers, writes about it plugins:

> *Still, in the case of popular open-source projects, I believe using a modular architecture offers some very strong advantages that go*

*far beyond the idea that people may have in mind when thinking about plugins. More than just a way to open your project to new features, they also provide crucial structure and support, helping those projects to stand the test of time[20].*

### 2.2.7 MINIX Operating System

MINIX was developed by Andrew S. Tanenbaum and initially positioned as an educational operating system that implements a microkernel architecture. The goal was also to create an operating system with high availability, fault tolerance, and comprehensibility, at a time when other operating systems contained millions of lines of code in C/C++, compiled as a monolithic kernel where approximately 70% of the code consisted of device drivers written by third-party developers, where a bug in one of the millions of lines could render the entire system inoperable. Tanenbaum saw this as a problem, namely a problem in the architectural approach. As a result, the MINIX 3 kernel was able to achieve 4000 lines of code in the kernel, with the remaining functionality moved to user-mode processes with restrictions on what they could do and how they could communicate with other processes. Interestingly, without MINIX, Linux might not have emerged[9].

### 2.2.8 Summary

Summarizing, the examples provided demonstrate the wide applicability of microkernel architecture across various domains such as software development, web applications, browsers, operating systems, and package managers. The advantages of microkernel architecture, including modularity, security, reliability, and adaptability, make it a valuable choice for critical applications. The examples cited above showcase the broad benefits of microkernel architecture in diverse fields, underscoring its versatility and applicability across different software development scenarios.

## 2.3 Analysis of Microkernel Architecture

The microkernel architecture, also referred to as a plugin architecture, has remained a prominent approach to software development for several decades[6]. Initially coined in the domain of operating systems, the term "microkernel" denotes the strategy of minimizing the kernel's scope by delegating non-essential functionalities to user space through modules. The primary objectives encompass maintaining a lean core, delineating distinct areas of responsibility, ensuring modularity, bolstering fault tolerance, and facilitating system extensibility[6, 9, 21].

When scrutinizing microkernel architecture in applications beyond the operating system realm, it's crucial to acknowledge that the prefix "micro" can be somewhat misleading since it doesn't always imply a diminutive size.

This investigation predominantly delves into microkernel architecture within higher-level applications, distinct from operating systems.

Four cardinal principles can be delineated for microkernel architecture[6, 9, 21]:

1. **Minimal Core:** Striving for a compact core that handles essential functions, relegating non-critical tasks to external modules.
2. **Separation of Responsibilities:** Demarcating distinct functional domains within the system, enhancing clarity and maintainability.
3. **Modularity:** Promoting a modular design where functionalities are encapsulated within independent units, facilitating flexibility and ease of development.
4. **Extensibility and Fault Tolerance:** Ensuring the system's ability to accommodate future enhancements seamlessly while resiliently handling errors and failures.

### 2.3.1 Topology

In the context of applications beyond the operating system level, microkernel architecture is a relatively simple monolithic structure comprising two key architectural components: the core and plug-in components[6]. The general topology of microkernel architecture is depicted in **Figure 2.3**:

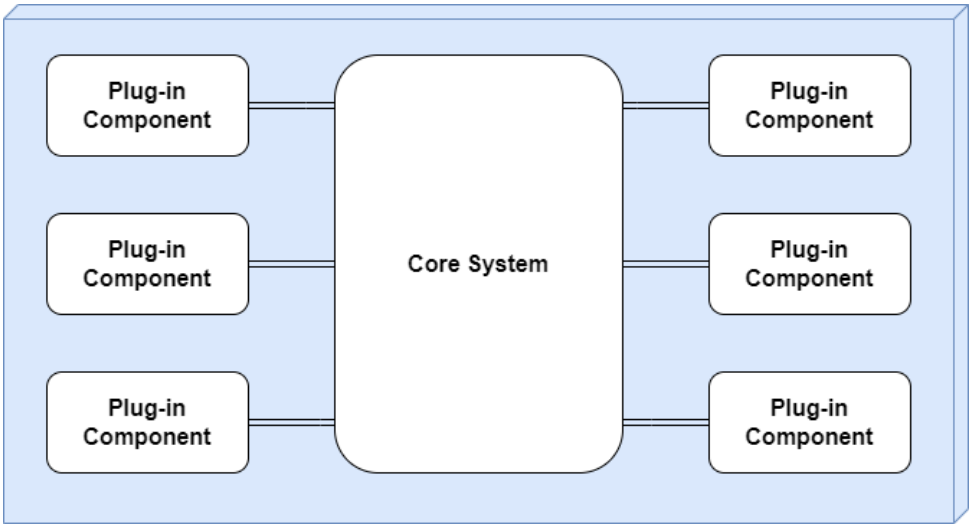



**Figure 2.3:** Basic components of the microkernel architecture[6].

### 2.3.2 Core System

At its core system, the microkernel architecture defines the basic system functionality, often described as the minimum requirements for initiating system operations[6, 9, 21]. For instance, the Eclipse IDE has a core system that is similar to a basic text editor, capable of opening, modifying, and saving files. However, it is the integration of plug-ins that transforms Eclipse into a fully functional and practical product[6, 11].

An alternative interpretation of the core system is to define the application's general processing flow or "happy path[2]", which is characterised by minimal or no custom processing. This approach involves extracting the cyclomatic complexity from the core system and relocating it to distinct plug-in components. This strategy improves the system's extensibility, maintainability, and testability[6].

The foundational system can be structured using either a layered architecture or a modular monolith, depending on its size and complexity[6]. **Figure 2.4** depicts the layered separation.

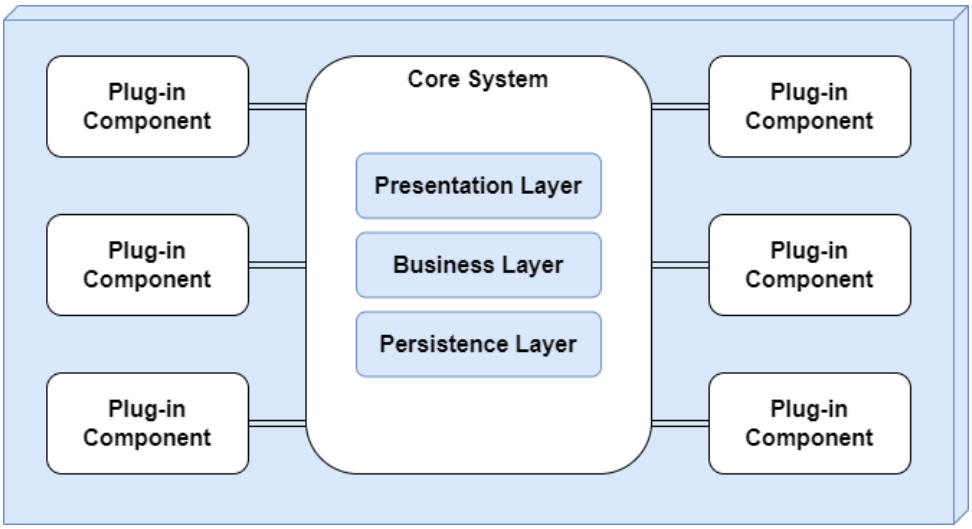



**Figure 2.4:** Layered core system (Technically partitioned)[6].

[2] "Happy path" is a term used in software testing that denotes the primary, most typical, and successful path of executing a program or functionality without encountering any issues or errors.

Alternatively, the core system may be partitioned into distinct, independently deployed domain services, as depicted in **Figure 2.5**. Each of these domain services would encompass specific plug-in components tailored to its particular domain[6].

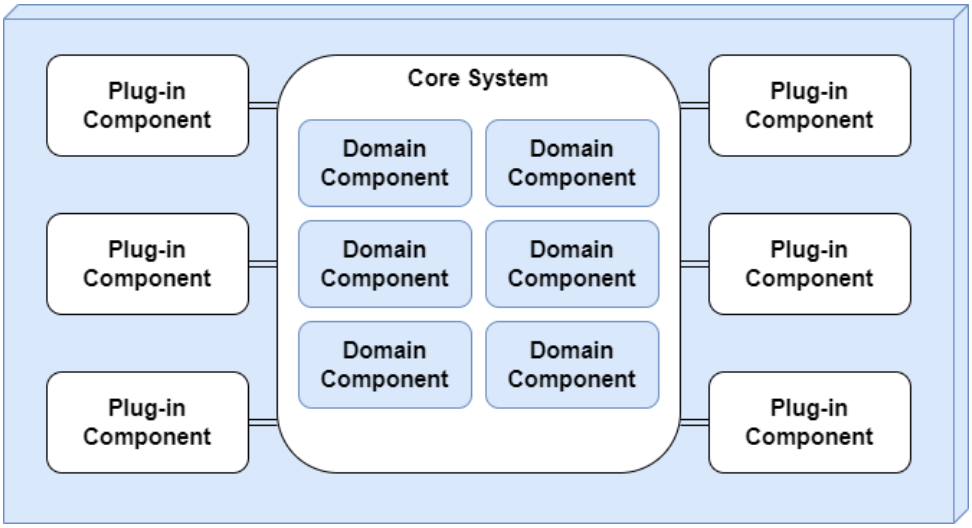



**Figure 2.5:** Modular core system (Domain partitioned)[6].

If an application includes a user interface, the presentation layer can be integrated into the core, be an independent component, or even implement a microkernel architecture for potential extension through plug-ins. All three scenarios are depicted in **Figure 2.6**, **Figure 2.7** and **Figure 2.8**[6].

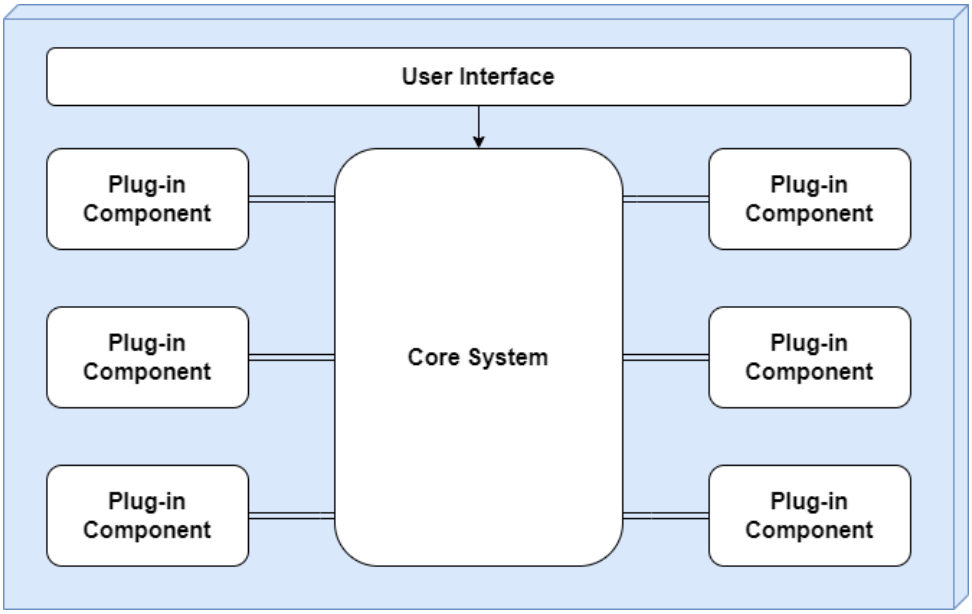



**Figure 2.6:** Embedded user interface (Single deployment)[6].

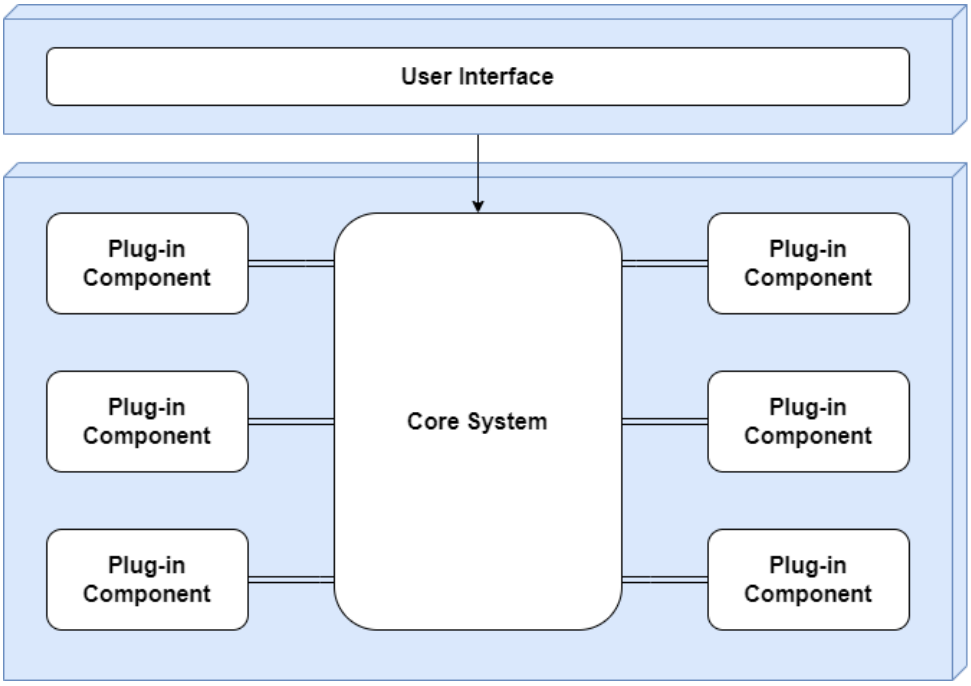



**Figure 2.7:** Separate user interface (Multiple deployment units)[6].

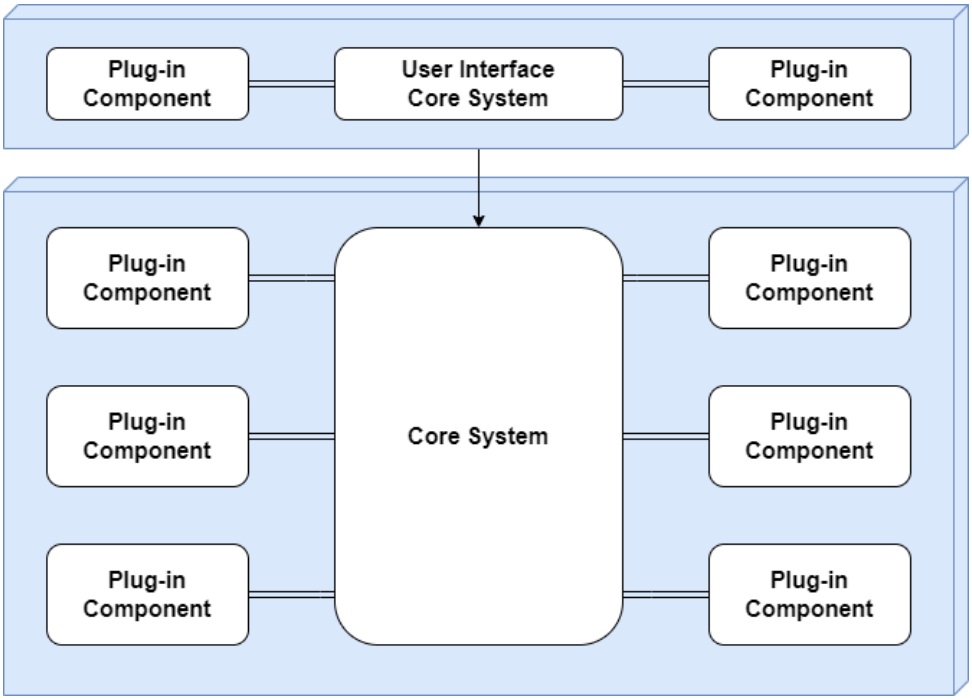



**Figure 2.8:** Separate user interface (Multiple deployment units, both Microkernel)[6].

### 2.3.3 Plug-in Components

Plug-in components serve as autonomous entities, housing specialised processing, additional features, and custom code designed to augment or extend the core system. They play a crucial role in isolating volatile code, thereby enhancing the maintainability and testability of the application. An ideal characteristic of plug-in components is their independence from each other, free from interdependencies[6, 21].

Communication between plug-in components and the core system typically follows a point-to-point model, where the connection resembles a method invocation or function call to the entry-point class of the plug-in. It is also quite common for communication to be realised using an event-based model, where plugins subscribe and react to certain events in the system. Furthermore, plug-in components can exist in either compile-based or runtime-based forms. Runtime plug-ins offer the advantage of dynamic addition or removal without necessitating the redeployment of the core system or other plug-ins. On the other hand, compile-based plug-in components, though simpler to manage, require redeployment of the entire monolithic application when modified, added, or removed[6].

During point-to-point communication, components are typically represented as shared libraries (Java archive (JAR), Dynamic-link library (DLL), Gem), package names in Java, or namespaces in other languages. **Figure 2.9** illustrates an example utilising JAR archives[6].

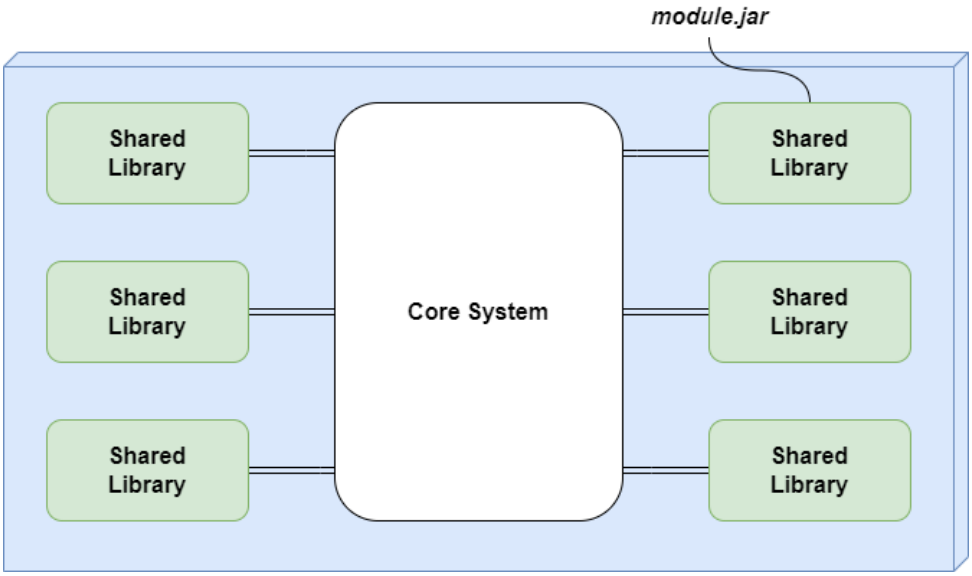



**Figure 2.9:** Shared library plug-in implementation[6].

One of the simplest ways to organise components is by using packages or namespaces within a single codebase, as illustrated in **Figure 2.10**. When adopting such a component organisation, careful consideration should be given to the naming conventions for packages or namespaces. One recommended approach involves employing the following template for naming packages

and namespaces: `app.plugin.<domain>.<context>`. The second element (plugin) facilitates the clear identification of the component as a plugin, the third component (domain) corresponds to the component's domain, and the fourth element (context) specifies the particular plugin[6].

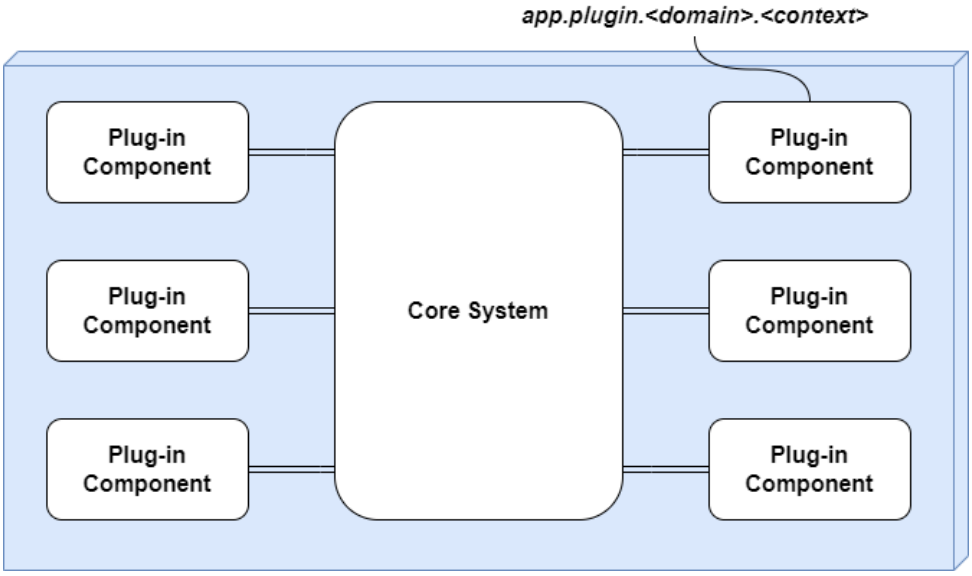



**Figure 2.10:** Package or namespace plug-in implementation[6].

Plug-in components do not always communicate with the core through point-to-point; communication can also be remote, facilitated by REST requests or messages. In such cases, each plugin component serves as a service, communicating remotely with the core. While this approach may seem to increase overall scalability, it is not the case; we still have a microkernel architecture with its monolithic core through which all communication passes (illustrated in **Figure 2.11**)[6].

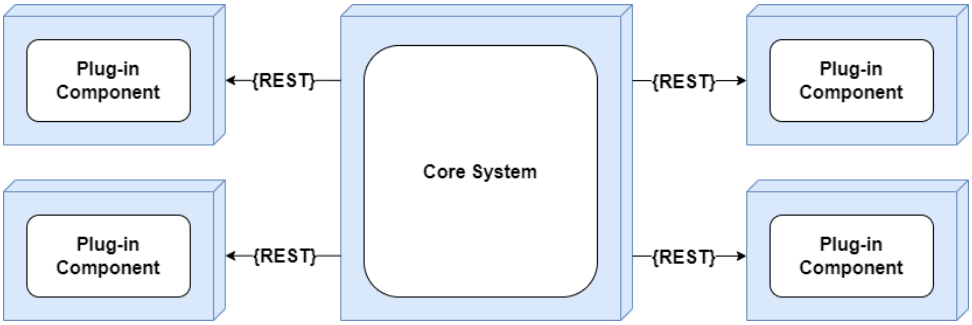



**Figure 2.11:** Remote plug-in access using REST[6].

The utilisation of a remote access approach for accessing plug-in components implemented as distinct services offers several advantages. It enhances overall component decoupling, affording improved scalability and throughput. Furthermore, this approach enables runtime modifications without reliance on specialised frameworks such as OSGi (used in Eclipse IDE), Jigsaw, or

Prism. Additionally, it facilitates asynchronous communication with plug-ins, potentially leading to a substantial enhancement in overall user responsiveness, contingent upon the specific scenario[6].

Despite the advantages of this approach, it has explicit drawbacks. Remote plugin connection shifts the microkernel architecture from the realm of monoliths to the domain of distributed architectures, making it more complex for development, testing and deployment. Remote communication also adds cost to the entire system, complicating the final deployment topology. Additionally, this approach introduces communication-related issues. If a plugin fails to work or is inaccessible due to network problems, the request simply won't be executed, a scenario that cannot occur in a monolithic deployment[6]. However, there are business cases where this is necessary. One such case is Shopify, a service whose business is largely built on plugins that extend its functionality and relies on third-party developers deploying plugins to developer servers. Another example is GitHub Copilot product, which is implemented as a plugin for many text editors and IDEs, with the core functionality, the Artificial Intelligence (AI) itself, implemented on a remote server.

Regarding data storage, it is not common in microkernel architecture for plugins to have a shared repository. Typically, each component of the system, including the system core, has its own storage. If there is a need to share data between different plugins, the system core usually takes responsibility for this. The main goal of this approach is to reduce coupling between plugins. **Figure 2.12** illustrates this approach[6].

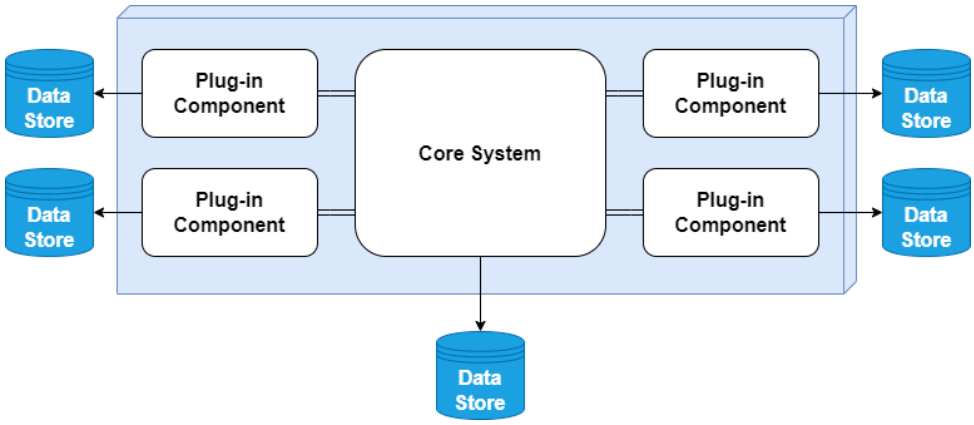



**Figure 2.12:** Plugin and core data storage[6].

### 2.3.4 Registry

The core needs to be aware of connected plugins and how it can obtain information about them. One commonly used solution for this is the use of a plugin registry component[6, 21]. The registry stores information about each connected plugin, including details such as the name, contract information (input and output data), and communication protocol information (if plugins

are connected remotely)[6].

This component can be a part of the core or external. Ready-made comprehensive solutions can also serve as plugin registries, and some examples include etcd, Eureka, Zookeeper, and Consul.

Internally, the plugin registry can be a simple hash map, where the key is the hash or name of the plugin, and the value is a reference to the connected component[6]. Alternatively, it can be a complex solution with registry and discovery functionality, similar to the ones mentioned above.

The basic sequence diagram for adding a new plugin through the user interface is presented in **Figure 2.13**.

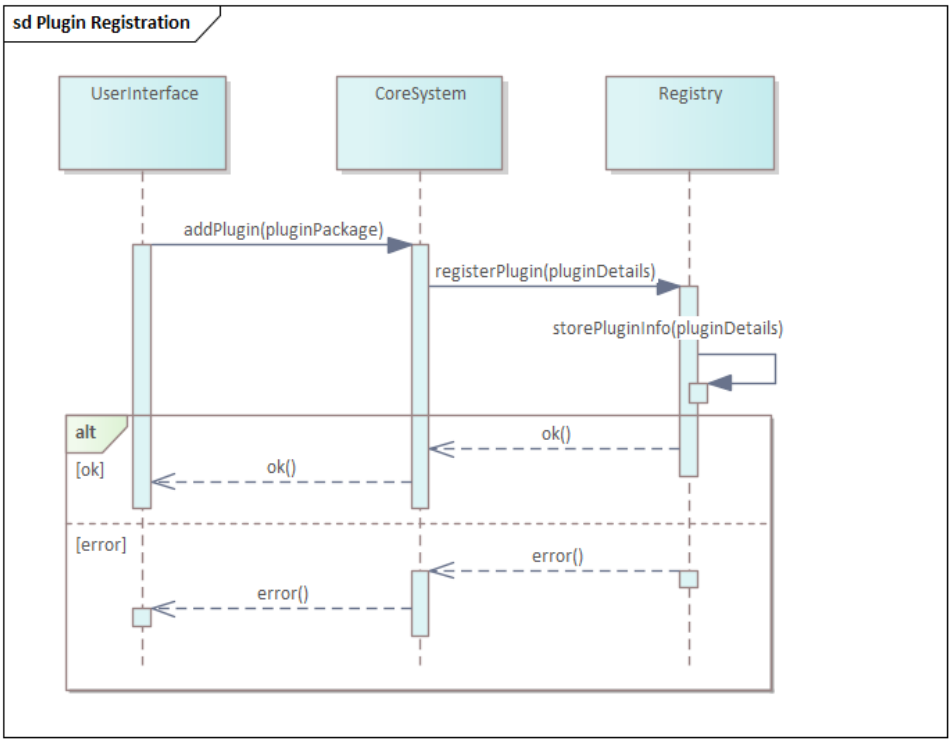



**Figure 2.13:** Sequence diagram of adding a new plug-in.

### 2.3.5 Contracts

In the realm of software development, contracts serve as essential components for delineating integration points within an architectural framework. These contracts manifest in various formats and play integral roles throughout the software design process. In essence, a contract can be defined as the standardized structure utilized by different architectural components to communicate crucial information and dependencies[22].

One of the fundamental concepts of a microkernel architecture is the use of contracts between plugin components and the system's core. These contracts represent an agreement on input and output data during communication between the core and plugin components and often relate to specific domains

or functionalities within the core. It is assumed that only standard contracts, which are strictly defined in the system, are used. If a plugin component has been developed by third-party developers, it may have a custom contract. In such cases, an adapter can be used without modifying the third-party component[6].

Contracts can be implemented in various ways, such as using Extensible Markup Language (XML), JavaScript Object Notation (JSON), Protocol Buffers, Data Transfer Objects (DTOs), or interfaces[6, 22].

### 2.3.6 Extension Points

Another crucial concept in microkernel architecture is Extension Points. Extension Points allow plug-ins to inject their logic into various parts of the system and alter its behaviour. Access to extension points is typically provided through the system core's API.

The documentation for the Eclipse IDE provides an excellent metaphor describing extensions and extension points:

> *The simplest metaphor for describing extensions and extension points is electrical outlets. The outlet, or socket, is the extension point; the plug, or light bulb that connects to it, the extension[23].*

Implementations of this concept can vary depending on the product. For example, in the Eclipse IDE, extension points define a contract, usually a combination of XML and a Java interface, to which extensions must adhere. If a plugin wants to connect to a specific extension point, it needs to implement this contract in its extension. The Eclipse IDE also allows plugins to define their own extension points, enabling them to extend each other's functionality[23].

In Eclipse IDE, extending extension points typically occurs through the use of extensions and extension points in the `plugin.xml` file[23].

**Listing 2.1** demonstrates the extension of the standard extension point `org.eclipse.ui.menus`[24].

**Listing 2.1:** Extension for the standard extension point `org.eclipse.ui.menus`.

```
<extension id="add.item" point="org.eclipse.ui.menus">
    <menuContribution
   locationURI="menu:someorg.somemenu.id?after=additions">
        <command
            commandId="someorg.someid.someCommand"
            icon="icons/anything.png"
            id="someorg.someid.BasicCmdItem"
            label="Simple Item"
            mnemonic="S">
        </command>
    </menuContribution>
</extension>
```

In **Listing 2.2**, the definition of an internal extension for `org.eclipse.ui.editors` is presented[25].

**Listing 2.2:** Internal editor extension definition for `org.eclipse.ui.editors`.

```
<extension point="org.eclipse.ui.editors">
    <editor
       id="com.xyz.XMLEditor"
       name="Fancy XYZ XML editor"
       icon="./icons/XMLEditor.png"
       extensions="xml"
       class="com.xyz.XMLEditor"
       contributorClass="com.xyz.XMLEditorContributor"
       symbolicFontName="org.eclipse.jface.textfont"
       default="false">
    </editor>
</extension>
```

In WordPress, for example, extension points are implemented using hooks. WordPress hooks come in two types: action and filter. To use actions and filters, we need to define a callback and then register it using the WordPress hook for a specific action or filter[14, 15].

Actions are used to perform specific tasks without taking any input or returning any values. On the other hand, filters are used to modify data. A filter-type hook takes data as input and returns modified data[14, 15].

Similar to Eclipse IDE, WordPress has predefined extension points and also allows plugins to create their own extension points.

In **Listing 2.3** and **Listing 2.4**, examples of action and filter registration in WordPress are presented[26, 27].

**Listing 2.3:** WordPress action example.

```
function wporg_callback() {
    // do something
}
add_action( 'init', 'wporg_callback' );
```

**Listing 2.4:** WordPress filter example.

```
function wporg_filter_title( $title ) {
    return 'The ' . $title . ' was filtered';
}
add_filter( 'the_title', 'wporg_filter_title' );
```

The functions **add_action** and **add_filter** can also take priority as the third parameter, influencing the order of execution of actions and filters[26, 27].

# Chapter 3

# Prototype

In this chapter, we will implement a proof-of-concept prototype of an application using a microkernel architecture to demonstrate its effectiveness in developing applications that require maximum flexibility in extending functionality, which, as we defined in **Section 2.1.2**, is its strength.

In order to illustrate the advantages of the microkernel architecture, it was determined that an engine for task automation "AutomationWizard" should be implemented. The user will be able to create and execute scenarios based on the functionality provided by third-party plugins. Given that scenarios and tasks can differ, the microkernel architecture approach appears to be the most suitable in this context and will enable the potential of the plugin architecture to be realised.

Next, to begin with, we will explore existing solutions for task automation, which can assist us in designing the prototype.

## 3.1 Existing Solutions

There are off-the-shelf solutions for task automation that also apply plugin architecture. These solutions provide the user with the ability to create and execute scenarios using third-party plugins, making them flexible and customisable to meet specific needs. Let's take a look at some of them to identify the benefits and features of plugin architecture in the context of task automation.

### 3.1.1 Make.com

Make.com is a web service that facilitates the automation of business processes through the use of no-code automation. Users can create scenarios using pre-built blocks and execute them on a schedule, manually, using custom webhooks, or based on events triggered by specific blocks (apps). Additionally, Make.com provides a convenient method for users to create custom blocks and modules by configuring them in JSON through a user-friendly interface. Make.com provides its own markup language, mustache-like in appearance, which enables the execution of JavaScript expressions within a sandbox

environment. This language also includes predefined functions and variables, which collectively constitute a Domain-Specific Language (DSL)[28].

### 3.1.2 Zapier

Zapier is another web service for automating business processes. It is similar to Make.com, but at the time of writing this work, it only supports extensions in beta mode and in the context of creating custom actions for existing applications[29].

### 3.1.3 Hookdeck

Hookdeck is a web service for the management of webhooks. It enables users to configure the flow between webhooks in a flexible manner, to create their own filters and transformations described in JSON and JavaScript, and to run them in a sandbox[30]. In comparison to Make.com and Zapier, Hookdeck provides a lower-level abstraction, whereas Make.com and Zapier essentially wrap webhooks in blocks.

## 3.2 Architecture

Within this prototype, a microkernel architecture with remote plugins will be implemented, where all communication between the core and plugins will be based on REST API and webhooks. This approach was mentioned in **Section 2.3.3**. Compared to the mentioned services Make.com and Hookdeck, our prototype will be something in between. We will also utilize webhooks for asynchronous communication at its core. Plugins will implement the necessary interfaces and contracts for interaction with the core and provide their extension points in the form of actions and triggers, which essentially will also be webhooks. Such an approach will allow plugin developers to develop them in any language, using any technologies and deployment environments, as long as they implement a predefined interface for interaction with the core.

## 3.3 Technologies

The system core is implemented in Java using the Spring Boot framework. Since in the context of the prototype, the core communicates with plugins via REST API and no serious computations are performed on the core side, Spring Boot is the optimal choice. Hibernate is used for persistence, and Maven is used for dependency management. The embedded H2 database was chosen as the database because it does not require additional configuration and installation, simplifying the deployment of the prototype. Nashorn is used to run user JavaScript in a sandbox for data filtering, transformation and controlling scenario execution. Quartz is used to implement the ability to schedule scenarios, as it allows for job persistence and dynamic modification.

Additionally, Spring WebFlux is used as the Hypertext Transfer Protocol (HTTP) client for asynchronous request sending. For basic authentication and authorization, Spring Security is used with JSON Web Token (JWT) tokens.

## 3.4 Design

The core of the prototype is a multi-layered application where layers are separated technically rather than by domains, as previously discussed in **Section 2.3.2**. This approach is typical when developing REST services with Spring Boot. Given that the prototype's communication is essentially based on REST APIs and webhooks, such separation of layers is optimal.

**Figure 3.1** represents a class diagram showing the entities in our prototype, their attributes, and the relationships between them.

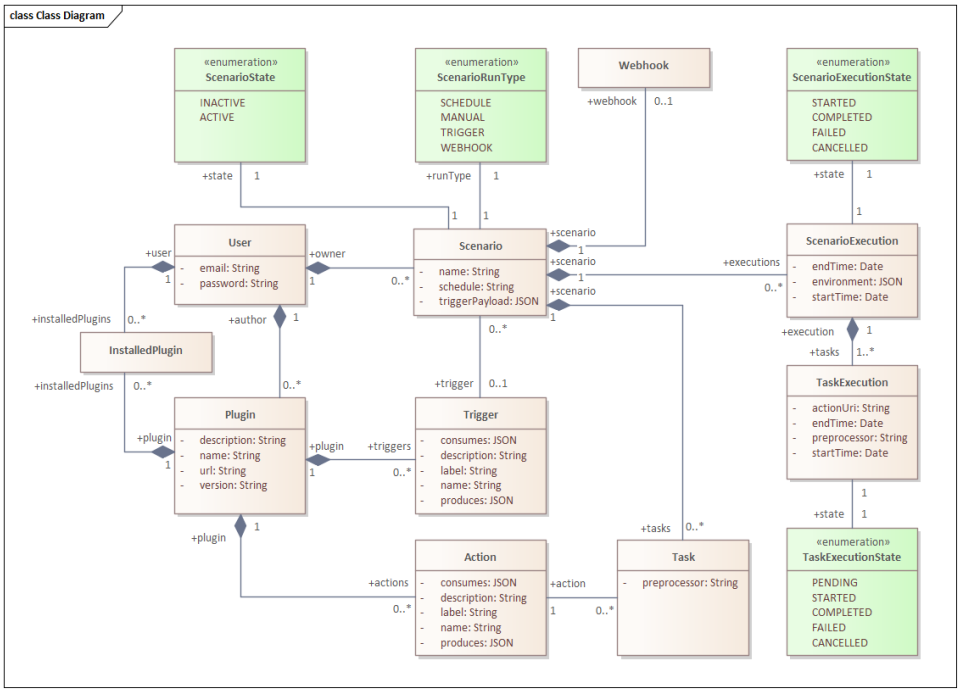



**Figure 3.1:** Prototype class diagram.

Let's break down the entities and their relationships a bit further. The `Plugin` entity represents, as the name suggests, the plugin itself and basic information about it. Plugins in the system are registered by users, so each plugin has an author. Upon registration, a plugin also registers triggers and actions, represented in the diagram as `Trigger` and `Action` entities. Triggers represent events of the plugin to which the core can subscribe for subsequent scenario execution, while actions are actions provided by the plugin to be executed within the context of a scenario.

The `Scenario` entity stores basic information about the scenario, its status, launch type (scheduled, manual, triggered by plugin event, or user event

through a custom webhook), trigger payload (in case of plugin trigger launch type) as well as the sequence of tasks required to be executed within the scenario context.

The `Task` entity references a plugin action that needs to be executed, as well as user JavaScript in Base64 for execution in a sandbox. This is necessary to allow the user to modify the payload between tasks, filter it, or terminate scenario execution based on certain conditions.

The `ScenarioExecution` and `TaskExecution` entities store information about the context of a specific scenario execution. This is necessary to avoid a tight coupling between execution and configuration, allowing configuration changes to be made to a scenario without affecting already launched scenarios with the old configuration. Additionally, `ScenarioExecution` has an environment field where the common environment for all executed tasks within the scenario execution is stored, allowing data sharing between tasks.

The deployment of the application is quite simple and intentionally simplified within the scope of the prototype. The deployment diagram is shown in **Figure 3.2**. In a real-world application, deployment would be more complex and would involve components such as load balancers, caches, databases, etc.

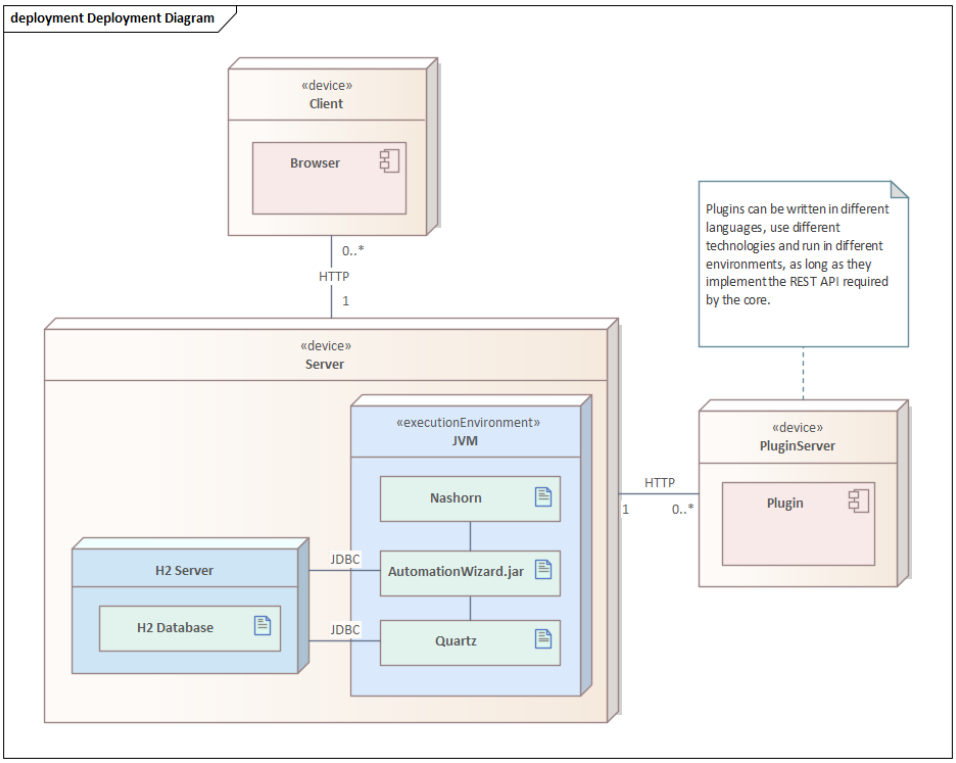



**Figure 3.2:** Prototype deployment diagram.

As mentioned earlier, the core and plugins communicate via REST API and webhooks, with each plugin having a base Uniform Resource Locator (URL) address. Let's review the existing endpoints and webhooks used in the interaction between the core and plugins. All paths will be specified relative

to the base URL address.

Let's consider the endpoints and webhooks that a plugin must implement to interact with the core:

- Path: /, Method: **POST**

  Returns information about the plugin, its actions, and triggers. This endpoint is used when the plugin registration webhook is triggered.

- Path: /, Method: **GET**

  Returns Hypertext Markup Language (HTML) for embedding on the plugin page. This endpoint is required to display the plugin's user interface in an iframe and is necessary to allow plugins to store additional information for each user, such as global plugin configuration, whether it's tokens, keys, or other data.

- Path: /, Method: **DELETE**

  This endpoint is used when the plugin deletion webhook is triggered.

- Path: **/install**, Method **POST**

  This endpoint is used when the plugin installation webhook is triggered by a user. It informs the plugin about its installation by a specific user.

- Path: **/uninstall**, Method: **POST**

  This endpoint is used when the plugin uninstallation webhook is triggered by a user. It informs the plugin about its uninstallation by a specific user.

- Path: **/triggers/{trigger-name}**, Method: **POST**

  This endpoint is used by the core to subscribe to the triggering of a specific plugin event.

- Path: **/triggers/{trigger-name}**, Method: **DELETE**

  This endpoint is used by the core to unsubscribe from the triggering of a specific plugin event.

- Path: **/actions/{action-name}**, Method: **POST**

  This endpoint is used by the core to send a request to execute a specific plugin action.

Now let's look at the endpoints and webhooks provided by the core for handling webhooks triggered by plugins and users:

- Path: **/api/v1/webhooks/tasks/{executionTaskId}**,
  Method: **POST**

  This endpoint is used by the core to receive a response about task execution from the plugin.

- Path: **/api/v1/webhooks/triggers/{scenarioId}**, Method: **POST**

  This endpoint is used by the core to handle an event that occurred on the plugin side for processing and triggering a scenario.

- Path: **/api/v1/webhooks/{webhookId}**, Method: **POST**

  This endpoint is used by the core to trigger a scenario based on a user event. It can be used in third-party services or applications.

The endpoints listed above essentially represent interfaces and contracts necessary for interaction between the core and plugins. Plugins must implement these interfaces and contracts for proper interaction with the core. In turn, the core should handle requests from plugins and users, as well as notify plugins of events occurring in the system. We will discuss their implementation in more detail in the **Section 3.5**.

For a better understanding, **Figure 3.3** depicts the sequence diagram in the context of triggering a scenario by a plugin and executing a scenario consisting of two tasks from two different plugins, where the asynchronous communication between the core and plugins is also visible.

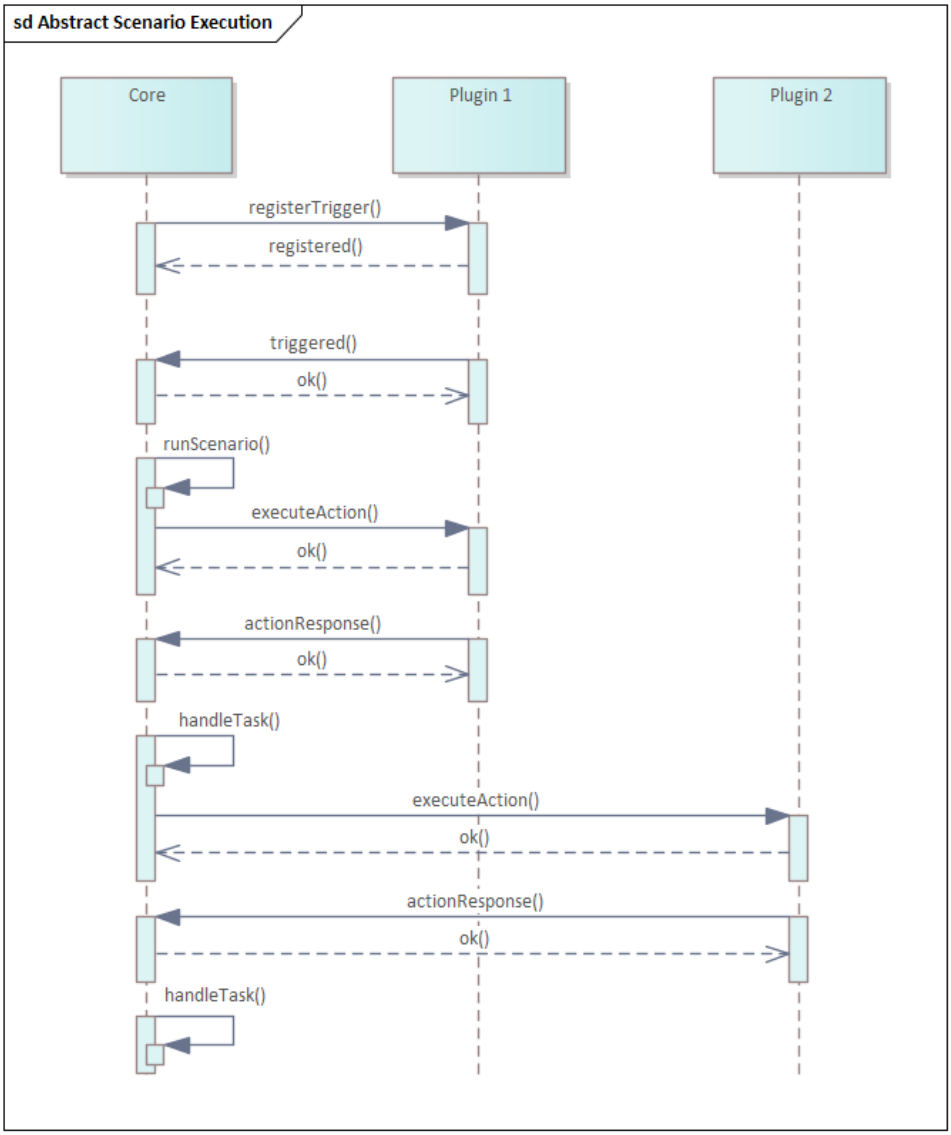



**Figure 3.3:** Abstract scenario execution sequence diagram.

## 3.5 Implementation

As mentioned in the **Section 3.3**, the Spring Boot framework was used to implement this prototype. The main Create, Read, Update, and Delete (CRUD) operations were implemented using the capabilities of this framework and are not particularly interesting within the scope of this work. In this section, we will take a closer look at contracts mentioned in the design section, examine how microkernel architecture components were implemented within this framework, and highlight the most interesting aspects of the prototype implementation.

### 3.5.1 Registry

The `Registry` component mentioned earlier in **Section 2.3.4** is implemented using Spring Boot Data JPA, allowing us to abstract away from the database logic and work with data at the object level, easily saving and retrieving it. The `Plugin` entity is mapped to the database table, and the `PluginRepository` interface is used to interact with the database. The `PluginService` class is used to implement business logic for working with plugins, such as registration, deletion, installation, and retrieval. The `PluginController` class is used to handle requests from users.

### 3.5.2 Contracts

Communication between the core and plugins is facilitated through REST API, as mentioned earlier in the **Section 3.4**. Next, we will delve deeper into the contracts that both plugins and the core must implement to interact with each other.

Let's delve into the contracts regarding the plugin endpoints:

- Path: /, Method: **POST**

  For successful registration, the plugin must return a JSON object in the response body with information about the plugin itself, its actions, and triggers. Example in **Listing 3.1**.

**Listing 3.1:** Response body with information about the plugin.

```
{
    "name": "Plugin name",
    "description": "Plugin description",
    "actions": [
        {
            "name": "action1",
            "label": "Action 1",
            "description": "Action 1 description",
            "consumes": {
                "$schema": "http://json-schema...",
```

```
                "title": "Message",
                "type": "object",
                "properties": {
                    "message": {
                        "type": "string"
                    }
                },
                "required": [
                    "message"
                ]
            },
            "produces": {}
        }
    ],
    "triggers": [
        {
            "name": "trigger1",
            "label": "Trigger 1",
            "description": "Trigger 1 description",
            "consumes": {},
            "produces": {}
        }
    ]
}
```

Actions and triggers must also contain information about the expected input and output data. For this, a contract described in JSON Schema is used. This is necessary for validating data in the context of creating scenarios from triggers and actions, which allows informing the user in advance that data transformation is required. Since standardizing the payload format for all actions and triggers would heavily limit plugin capabilities, in our prototype, we use Nashorn to allow users to transform and filter data using JavaScript, as well as influence the execution process itself.

- Path: `/`, Method: **GET**

  The plugin's user interface should be implemented on the plugin side, following a similar approach used by Shopify. When accessing this endpoint, the system core passes the user ID in the request header `X-User-Id`.

- Path: `/`, Method: **DELETE**

  When a plugin is deleted, the system core sends a request to this endpoint. The plugin should remove all user-specific data and configurations. Upon successful execution, the plugin should return a response with status `204 No Content`.

- Path: **/install**, Method **POST**

  The user ID is passed in the request header `X-User-Id`. This can be used by the plugin to store user-specific information if necessary.

- Path: **/uninstall**, Method: **POST**

  The user ID is passed in the request header `X-User-Id`. This can be used by the plugin to remove user-specific information if necessary.

- Path: **/triggers/{trigger-name}**, Method: **POST**

  The request body expects a JSON object defined in the trigger's JSON Schema in the consumes field, and the request headers `X-User-Id` and `X-Scenario-Id` pass the user ID and scenario ID respectively. In this case, the plugin only registers the trigger, and the trigger itself is triggered asynchronously.

- Path: **/triggers/{trigger-name}**, Method: **DELETE**

  The request headers `X-User-Id` and `X-Scenario-Id` pass the user ID and scenario ID respectively. Upon successful execution, the plugin should return a response with status code `204 No Content`.

- Path: **/actions/{action-name}**, Method: **POST**

  The request body expects a JSON object defined in the action's JSON Schema in the consumes field, and the request headers `X-User-Id` and `X-Task-Execution-Id` pass the user ID and task execution ID respectively. In this case, only a request for action execution is sent, and the execution itself occurs asynchronously.

Now, let's consider the contracts regarding the core endpoints:

- Path: **/api/v1/webhooks/tasks/{executionTaskId}**, Method: **POST**

  The request body should contain a JSON object with the task execution status, an error message in case of `FAILED` status, and the task execution result in case of `SUCCESS` status. An example of such an object is presented in **Listing 3.2**. The request parameters include the task execution identifier.

  **Listing 3.2:** Task execution response body.

```
{
    "state": "SUCCESS",
    "message": "",
    "result": {}
}
```

- Path: **/api/v1/webhooks/triggers/{scenarioId}**, Method: **POST**

  The request body contains a JSON object with event data. The request parameters include the scenario identifier.

- Path: **/api/v1/webhooks/{webhookId}**, Method: **POST**

  The request body contains a JSON object with event data. The request parameters include the webhook identifier.

### 3.5.3 Extension Points

Extension points are implemented using webhooks, as mentioned in **Section 3.5.2**. The core provides webhooks to trigger scenarios by plugin triggers or user events using a custom webhook, as well as to handle responses from plugins regarding action execution. Plugins provide webhooks to trigger actions when a specific task from the scenario is executed. Thus, plugins can influence the behavior of the core and other plugins within the context of a scenario. Additionally, the core provides the ability to insert custom JavaScript code into the scenario for data filtering, transformation, and controlling the scenario execution process, as mentioned in **Section 3.4**.

### 3.5.4 Scenarios

The class responsible for triggering the scenario, handling plugin responses, and initiating subsequent tasks in the scenario is `ScenarioExecutionService`, presented in **Listing 3.3**. Let's examine the most interesting aspects of its implementation.

**Listing 3.3:** ScenarioExecutionService implementation.

```
@Slf4j
@Service
@RequiredArgsConstructor
public class ScenarioExecutionService {
    ...
    @Async
    @Transactional
    public void runScenario(UUID scenarioId, JsonNode
    payload) {
        ...
    }

    @Async
    @Transactional
    public void taskHandler(UUID taskExecutionId,
    PluginTaskExecutionRequest request) {
        ...
    }

    private Mono<Void> executeTask(TaskExecution
    taskExecution, ScenarioExecution scenarioExecution,
    JsonNode payload){
        ...
    }

    private Consumer<? super Throwable>
    taskErrorHandler(UUID taskExecutionId, UUID
    scenarioExecutionId) {
        ...
    }
}
```

**Listing 3.4** presents the implementation of the `runScenario` method, which triggers the scenario. This method is annotated with `@Async`, allowing it to be executed asynchronously without blocking the main execution thread. Additionally, a `ScenarioExecution` entity and `TaskExecution` entities are created for each task in the scenario. This approach ensures independence from the initial scenario configuration if it changes during scenario execution. Next, the first task of the scenario is executed using the `executeTask` method.

**Listing 3.4:** ScenarioExecutionService runScenario method implementation.

```
@Async
@Transactional
public void runScenario(UUID scenarioId, JsonNode payload) {
    ...
    // Create scenario execution
    final ScenarioExecution scenarioExecution =
    ScenarioExecution.builder()
            .scenario(scenario)

    .environment(JsonNodeFactory.instance.objectNode())
            .state(ScenarioExecutionState.STARTED)
            .build();

    // Create task executions
    var executionTasks = tasks.stream().map(task ->
    TaskExecution.builder()
            .state(TaskExecutionState.PENDING)
            .actionUri(task.getAction().getPlugin().getUrl()
    + "/actions/" + task.getAction().getName())
            .scenarioExecution(scenarioExecution)
            .preprocessor(task.getPreprocessor())
            .build()).toList();

    scenarioExecution.setTasks(executionTasks);
    scenarioExecutionRepository.save(scenarioExecution);

    // Execute first task
    executeTask(executionTasks.get(0), scenarioExecution,
    payload)
            .publishOn(Schedulers.boundedElastic())

    .doOnError(taskErrorHandler(executionTasks.get(0).getId(),
    scenarioExecution.getId()))
            .subscribe();
}
```

**Listing 3.5** provides the implementation of the `executeTask` method. In this method, the environment for the custom JavaScript is set, the custom script is executed in the sandbox, and the execution result is obtained for further processing. Then, an asynchronous request is sent to the plugin for action execution using the WebClient.

**Listing 3.5:** ScenarioExecutionService executeTask method implementation.

```
private Mono<Void> executeTask(TaskExecution taskExecution,
    ScenarioExecution scenarioExecution, JsonNode payload) {
    ...
    var stringBuilder = new StringBuilder();
    stringBuilder.append("var env = JSON.parse(env);");
    var script = new String(Base64.getDecoder().decode(
        taskExecution.getPreprocessor()));
    stringBuilder.append(script);
    stringBuilder.append("env = JSON.stringify(env);");
    script = stringBuilder.toString();
    boolean process = true;

    // Setup context
    SandboxScriptContext sandboxScriptContext =
    sandbox.createScriptContext();
    ScriptContext context =
    sandboxScriptContext.getContext();

    var environment = (ObjectNode)
    scenarioExecution.getEnvironment();
    environment.set("payload", payload);
    environment.set("process",
    JsonNodeFactory.instance.booleanNode(process));

    context.setAttribute("env", environment.toString(),
    ScriptContext.ENGINE_SCOPE);

    // Execute script
    try {
        sandbox.eval(script, sandboxScriptContext);
        var updatedEnvironment = new
    ObjectMapper().readTree((String)
    context.getAttribute("env"));
    ...
    return webClientBuilder.build().post()
        .uri(taskExecution.getActionUri())
        .contentType(MediaType.APPLICATION_JSON)
        .header("X-User-ID", scenarioExecution.getScenario()
        .getOwner().getId().toString())
        .header("X-Task-Execution-ID",
    taskExecution.getId().toString())
        .bodyValue(payload)
        .retrieve()
        .bodyToMono(Void.class);
}
```

**Listing 3.6** presents the implementation of the `taskHandler` method, which handles the response from the plugin regarding action execution. In case of successful action execution, it checks if there is a next task to execute. If there is, the `executeTask` method is called for its execution. In case of an error, the task status is set to `FAILED`, and the scenario execution is terminated with a `FAILED` status. The method is also annotated with `@Async` for asynchronous execution to prevent blocking the main execution thread.

**Listing 3.6:** ScenarioExecutionService taskHandler method implementation.

```
@Async
@Transactional
public void taskHandler(UUID taskExecutionId,
    PluginTaskExecutionRequest request) {
     ...
     // Find current task index
     var currentTaskIndex =
    executionTasks.indexOf(executionTask);
     if (currentTaskIndex == -1) {
         throw new NotFoundException("Task execution not
    found");
     }
     ...
     if (request.getState() ==
    PluginTaskExecutionRequest.PluginTaskExecutionState.FAILED)
    {
         executionTask.setState(TaskExecutionState.FAILED);

    scenarioExecution.setState(ScenarioExecutionState.FAILED);
         scenarioExecution.setEndTime(LocalDateTime.now());
         scenarioExecutionRepository.save(scenarioExecution);
         return;
     }
     ...
     // Check if task is last and complete scenario execution
     if (currentTaskIndex == executionTasks.size() - 1) {
         scenarioExecution.setState(
             ScenarioExecutionState.COMPLETED);
         scenarioExecution.setEndTime(LocalDateTime.now());
         scenarioExecutionRepository.save(scenarioExecution);
         return;
     }

     // Execute next task
     executeTask(executionTasks.get(currentTaskIndex + 1),
    scenarioExecution, request.getResult())
             .publishOn(Schedulers.boundedElastic())

    .doOnError(taskErrorHandler(executionTask.getId(),
    scenarioExecution.getId()))
             .subscribe();
}
```

In the next section, we will discuss the testing and evaluation of the prototype.

## 3.6 Testing and Evaluation

To test our prototype, we need to define user scenarios, plugins used, their involvement within specific user scenarios, and testing scenarios. Let's take a closer look at two usage scenarios:

- **Scenario 1**

  Upon the creation of a new task in a Jira project, it triggers the assignment of an assignee to the task and sends notifications to both the assignee and the project manager via Slack.

- **Scenario 2**

  Regularly, on a timer basis, it checks the price of a product in an online store. If the price drops below a certain threshold, it sends a notification to the user via Short Message Service (SMS).

Based on the described scenarios, the following plugins have been identified:

- **Jira**

  This plugin is a Jira integration providing capabilities to work with projects and tasks. It includes two actions: "Get Project Participants" and "Assign Task", as well as one trigger "New Task". Actions allow retrieving a list of project participants and assigning a task to a specific user, while the trigger activates upon the creation of a new task in the project.

- **Slack**

  This plugin serves as a Slack integration primarily designed to send private messages. It features a single action called "Send Private Message", allowing the transmission of messages to specific users on Slack. There are no triggers included in this plugin.

- **HTML**

  This plugin allows fetching HTML content from a specified URL. Its single action, "Get HTML", takes a URL as input and returns the corresponding HTML content. There are no triggers included in this plugin.

- **SMS**

  This plugin enables the sending of SMS messages. Its sole action, "Send SMS", accepts the phone number of the recipient and the message content as input. There are no triggers included in this plugin.

It's also worth mentioning that the plugins, their actions, and triggers mentioned above are just examples; in a real application, actions, triggers, and their parameters may vary. In our example, authorization is also omitted

for simplicity, but in a real-world application, if necessary, the plugin could prompt the user for access tokens to third-party services, either in the context of payload data or through plugin interface settings.

In **Table 3.1** and **Table 3.2**, definitions of integration tests for user scenarios 1 and 2 are provided, respectively, as previously outlined. They include the test identifier, test description, preconditions, input data, expected result, and postconditions.

**Table 3.1:** Definition of integration test Scenario 1.

| | |
|---|---|
| **ID** | Scenario 1 |
| **Description** | Integration test of core and plugins in the context of scenario execution triggered by a plugin. |
| **Precondition** | In the system, there exists a user. Jira and Slack plugins are registered, and the user has installed these plugins. A scenario with tasks is created and activated. |
| **Input** | A new task appears in the Jira project, triggering the plugin's trigger. |
| **Expected Outcome** | A notification about the new task in the Jira project successfully arrives in Slack to the assigned user for processing, as well as to the project manager. |
| **Postcondition** | ScenarioExecution has the status COMPLETED. TaskExecution of the executed scenario also has the status COMPLETED. |

**Table 3.2:** Definition of integration test Scenario 2.

| | |
|---|---|
| **ID** | Scenario 2 |
| **Description** | Integration test of core and plugins in the context of scenario execution triggered by a schedule. |
| **Precondition** | In the system, there exists a user. HTML and SMS plugins are registered, and the user has installed these plugins. A scenario with tasks is created and activated. |
| **Input** | The scheduler has been triggered. |
| **Expected Outcome** | SMS notification was successfully sent to the user when the condition is met. |
| **Postcondition** | ScenarioExecution has the status COMPLETED. TaskExecution of the executed scenario also has the status COMPLETED. |

The testing scenarios listed above represent the "happy path", aiming to verify that the system operates correctly under normal conditions. For our prototype and its goals, this level of testing suffices to demonstrate the system's functionality and potential. However, in the context of a production application, this would certainly be inadequate.

During the prototype development and testing process, certain testing

scenarios were conducted manually using Postman[1] and Mockoon[2]. However, for a prototype without a user interface, manual testing of such an application is quite laborious and not very convenient. Hence, automated integration tests were written.

To implement automated integration tests, the Spring Boot Starter Test and WireMock[3] packages were used for mocking plugins and testing interactions with them, including contract testing. It's worth noting that since communication between the core and plugins occurs asynchronously during the execution of a user scenario, testing such systems is a non-trivial task.

For a better understanding of the testing process of the previously described user scenarios, sequence diagrams illustrating the interaction between the core and plugins during scenario execution are provided in **Figure 3.4** and **Figure 3.5** respectively. These diagrams also indicate the payload and the applied user JavaScript code.

[1]Postman - a tool for testing APIs.
[2]Mockoon - a tool for mocking APIs.
[3]WireMock - a library for mocking APIs.

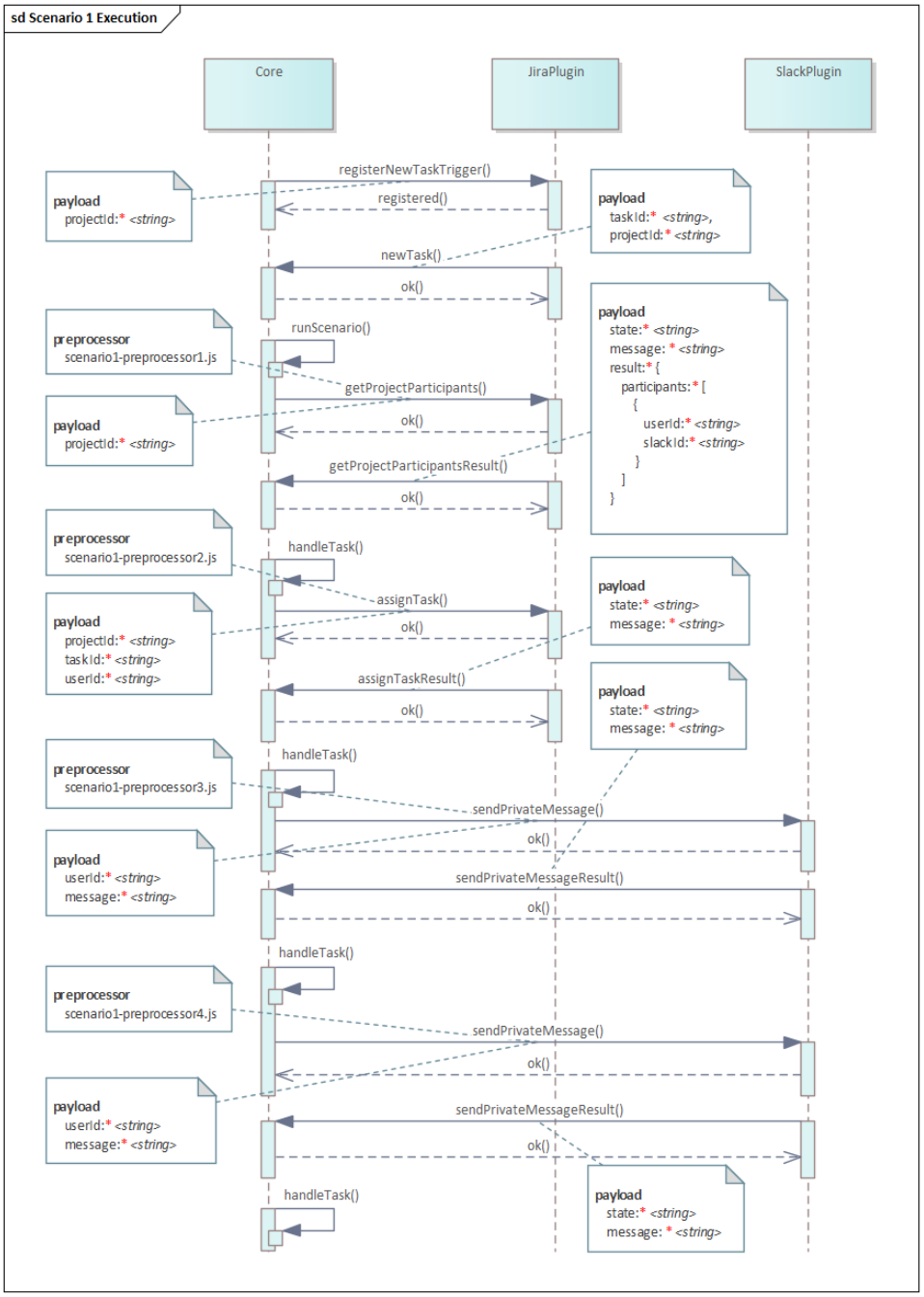



**Figure 3.4:** Scenario 1 execution sequence diagram.

Before processing the result of the executed plugin action, custom JavaScript code is executed in the sandbox for data transformation or filtering. Let's delve deeper into the custom JavaScript used in scenario 1.

**Listing 3.7** presents the custom JavaScript code for scenario 1, which is executed before performing the "Get Project Participants" action of the Jira plugin. To retrieve project participants, the action requires passing the project identifier obtained when the "New Task" trigger is triggered, which we set in the environment in the payload, where we also store the task identifier for use in subsequent tasks.

**Listing 3.7:** User's JavaScript for the Scenario 1, scenario1-preprocessor1.js

```
env.projectId = env.payload.projectId;
env.taskId = env.payload.taskId;
env.payload = {
    "projectId": env.projectId
}
```

**Listing 3.8** provides the custom JavaScript code for scenario 1, which is executed before performing the "Assign Task" action of the Jira plugin. In this code, we select a random project participant from the response of the "Get Project Participants" action, save their Slack identifier in the environment, and pass the project, task, and user identifiers in the payload.

**Listing 3.8:** User's JavaScript for the Scenario 1, scenario1-preprocessor2.js

```
if (env.payload.participants.length !== 0) {
    var randomIndex = Math.floor(Math.random() *
    env.payload.participants.length);
    var randomParticipant =
    env.payload.participants[randomIndex];

    env.slackId = randomParticipant.slackId;
    env.payload = {
        projectId: env.projectId,
        taskId: env.taskId,
        userId: randomParticipant.userId
    };
}
```

**Listing 3.9** depicts the custom JavaScript code for scenario 1, executed before performing the "Send Private Message" action of the Slack plugin. This code constructs a message to be sent to the project participant assigned the task.

**Listing 3.9:** User's JavaScript for the Scenario 1, scenario1-preprocessor3.js

```
env.payload = {
    "userId": env.slackId,
    "message": "You have been assigned a new task " +
   env.taskId + " in Jira project " + env.projectId
}
```

**Listing 3.10** presents the custom JavaScript code for scenario 1, executed before performing the "Send Private Message" action of the Slack plugin. In this code, a message is crafted to inform the project manager about the creation of a new task and its assignment to a project participant.

**Listing 3.10:** User's JavaScript for the Scenario 1, scenario1-preprocessor4.js

```
env.payload = {
    "userId": "9efbc1bf-0145-4f19-aa04-2d1337cb59da",
    "message": "A new task " + env.taskId + " was created in
   the project " + env.projectId + " and assigned to a user
   " + env.slackId
}
```

Let's take a closer look at the execution of the second scenario.

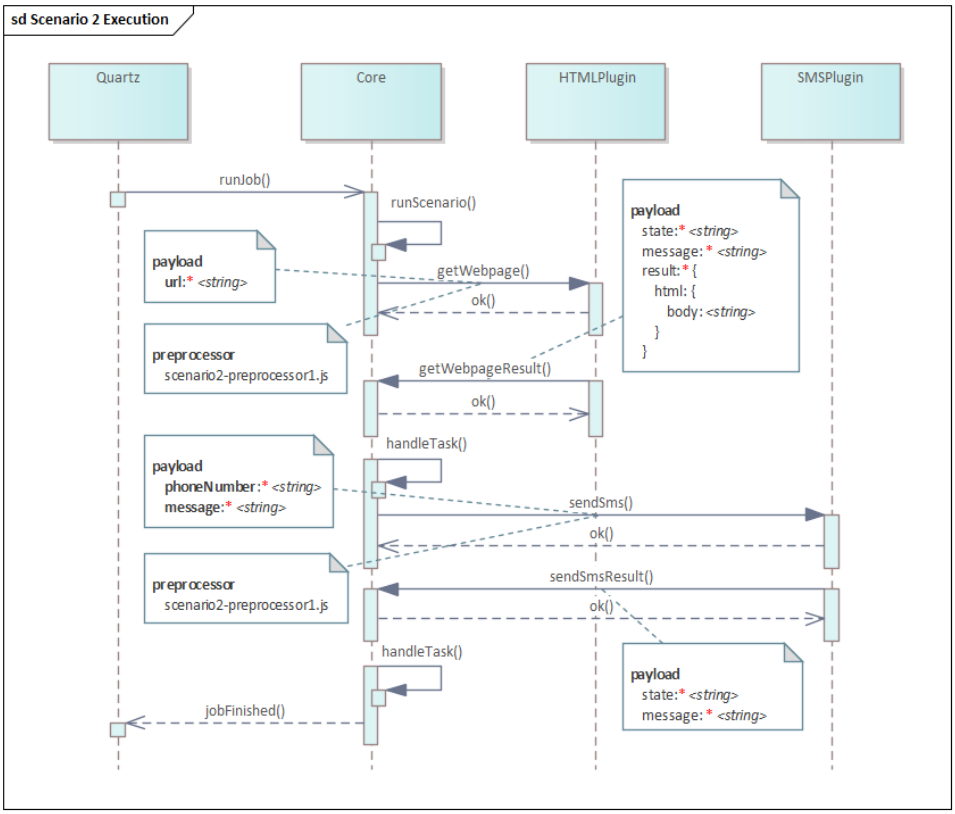



**Figure 3.5:** Scenario 2 execution sequence diagram.

**Listing 3.11** presents the custom JavaScript code for scenario 2, executed before performing the "Get HTML" action of the HTML plugin. This code constructs the payload for requesting HTML content from the specified URL.

**Listing 3.11:** User's JavaScript for the Scenario 2, scenario1-preprocessor1.js

```
env.payload = {
    "url": "https://test.com"
}
```

**Listing 3.12** provides the custom JavaScript code for scenario 2, executed before performing the "Send SMS" action of the SMS plugin. In this code, the price of the product is checked, and if it is less than 30, a message is crafted to send to the user; otherwise, the scenario execution is canceled.

**Listing 3.12:** User's JavaScript for the Scenario 2, scenario1-preprocessor2.js

```
if (env.payload.html.body < 30) {
    env.payload = {
        phoneNumber: '1234567890',
        message: 'Price is low, buy now!'
    };
} else {
    env.process = false;
}
```

These scenarios were successfully implemented and tested, allowing us to conclude on the prototype's functionality and its alignment with the set goals. Although the plugins in our example were relatively simple, our system does not limit their complexity and capabilities. Asynchronous communication enables plugins to execute tasks that require prolonged execution time, while custom JavaScript code allows users to influence the scenario execution process, filter, and transform data. Additionally, it's worth noting that in a real application, it's possible to implement non-linear scenario execution and support other types of transmitted data, such as binary data.

Authentication for plugins was not implemented in the prototype, which is essential in a production environment. Basic error handling was implemented in the prototype, but in a production, there would likely be a much larger number of edge cases to handle.

# Chapter 4

# Conclusion

Throughout this thesis, a comparison between the microkernel architecture and other types of architectures was conducted, successful use cases of this architecture in real-world applications were analyzed, and a detailed examination of its components and principles was performed. Subsequently, a prototype of a system using the microkernel architecture was implemented. The prototype was thoroughly tested and evaluated, leading to conclusions about its functionality and alignment with the set goals.

From all of this, it can be inferred that the microkernel architecture represents a promising approach to building scalable and flexible systems, enabling the development of applications that can be easily extended with new functionality without altering the original codebase. However, like any architecture, it involves trade-offs and priorities. Implementing a system based on the microkernel architecture requires careful attention to detail and meticulous planning, as incorrect system design can lead to issues that are difficult to rectify in production. This applies especially to the implementation of the core itself, where it is crucial to determine which functionality should be part of the core and which should be delegated to plugins. In the context of our prototype, the complexity is further compounded by the fact that all plugins are remote, which complicates testing and requires more focus on security and handling various edge cases.

It is hoped that this work will contribute to the popularization of the microkernel architecture, and we anticipate seeing more projects adopting this approach in the future.

The source code of the prototype core is available in the GitHub repository at the following link: `https://github.com/mikicit/automate-wizard-core`.

# Appendix A

# Bibliography


[1] Michael J. Accetta, Robert A. Baron, David B. Golub, Richard F. Rashid, and Avadis Tevanian. The past, present, and future for software architecture. *IEEE Software*, 23(2):22–30, March 2006. doi: 10.1109/MS.2006.59.

[2] William A. Wulf, Ellis Saul Cohen, William Corwin, Anita Katherine Jones, Roy Levin, C Pierson, and Fred Pollack. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6): 337–345, June 1974. doi: 10.1145/355616.364017.

[3] Richard F. Rashid. Accent: A communication oriented network operating system kernel. *ACM SIGOPS Operating Systems Review*, 15(5):64–75, December 1981. doi: 10.1145/1067627.806593.

[4] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022. doi: 10.1109/ACCESS.2022.3152803.

[5] James Lewis and Martin Fowler. Microservices: A definition of this new architectural term, March 2014. URL `https://www.martinfowler.com/articles/microservices.html`.

[6] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Inc., 2020. ISBN 978-1-492-04345-4.

[7] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153, 2020. doi: 10.1109/MEMSTECH49584.2020.9109514.

[8] Johannes Mayer, Ingo Melzer, and Franz Schweiggert. Lightweight plug-in-based application development. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 87–102, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36557-0.

[9] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Pearson Education, Inc., 2006. ISBN 0-13-142938-8.

[10] Mark Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, Inc., 2016. ISBN 978-1-491-95242-9.

[11] Eclipse Foundation. What is eclipse?, 2024. URL `https://www.eclipse.org/home/whatis/`.

[12] Richard M. Stallman. Emacs the extensible, customizable self-documenting display editor. In Paul Abrahams, editor, *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 147–156, New York, NY, United States, June 1981. Association for Computing Machinery. ISBN 978-0-89791-050-7.

[13] Inc. Shopify. Shopify apps overview, 2024. URL `https://shopify.dev/docs/apps/getting-started`.

[14] Brad Williams, Justin Tadlock, and John Jacoby. *Professional WordPress Plugin Development*. John Wiley & Sons, Inc, May 2020. ISBN 978-1-119-66694-3.

[15] Brad Williams, David Damstra, and Hal Stern. *Professional WordPress: Design and Development*. John Wiley & Sons, Inc, 2015. ISBN 978-1-118-98724-7.

[16] Inc. Google. Chromium, 2024. URL `https://www.chromium.org/Home/`.

[17] Inc. Google. Extensions manifesto, 2024. URL `https://www.chromium.org/developers/design-documents/extensions/how-the-extension-system-works/extension-manifesto/`.

[18] Yarnpkg. Yarn introduction, 2024. URL `https://yarnpkg.com/getting-started/`.

[19] Yarnpkg. Yarn plugin tutorial, 2024. URL `https://yarnpkg.com/advanced/plugin-tutorial`.

[20] Maël Nison. Plugin systems - when why, 2019. URL `https://dev.to/arcanis/plugin-systems-when-why-58pp`.

[21] Robert Chatley. *Predictable Dynamic Plugin Architectures*. Phd thesis, University of London, London, United Kingdom, February 2005. Available at `https://www.researchgate.net/profile/Robert-Chatley/publication/242434844_Predictable_Dynamic_Plugin_Architectures/links/55f2979f08aef559dc493b0e/Predictable-Dynamic-Plugin-Architectures.pdf`.

[22] Mark Richards, Neal Ford, Pramod Sadalage, and Dehghani Zhamak. *Software Architecture: The Hard Parts*. O'Reilly Media, Inc., 2021. ISBN 978-1-492-08689-5.

[23] Eclipse Foundation. Extensions and extension points, 2024. URL https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fconcepts%2Fextension.htm.

[24] Eclipse Foundation. Menus, 2024. URL https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ui_popupMenus.html.

[25] Eclipse Foundation. Editors, 2024. URL https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fextension-points%2Forg_eclipse_ui_popupMenus.html.

[26] WordPress Foundation. Actions – plugin handbook, 2024. URL https://developer.wordpress.org/plugins/hooks/actions/.

[27] WordPress Foundation. Filters – plugin handbook, 2024. URL https://developer.wordpress.org/plugins/hooks/actions/.

[28] Inc Celonis. Make, 2024. URL https://www.make.com/.

[29] Inc. Zapier. Zapier, 2024. URL https://zapier.com/.

[30] Inc. Hookdeck Technologies. Hookdeck, 2024. URL https://hookdeck.com/.

# Appendix B

## Acronyms

**AI** Artificial Intelligence. 31

**API** Application Programming Interface. 17, 33, 36–38, 42, 53

**CMS** Content Management System. 20

**CRUD** Create, Read, Update, and Delete. 42

**DLL** Dynamic-link library. 29

**DSL** Domain-Specific Language. 36

**DTO** Data Transfer Object. 33

**HTML** Hypertext Markup Language. 39, 51, 52, 57

**HTTP** Hypertext Transfer Protocol. 37

**IDE** Integrated Development Environment. 17, 31

**JAR** Java archive. 29

**JSON** JavaScript Object Notation. 33, 35, 36, 42–45

**JWT** JSON Web Token. 37

**RCP** Rich Client Platform. 17

**REST** Representational State Transfer. 30, 36–38, 42

**SEO** Search Engine Optimization. 20

**SMS** Short Message Service. 51, 52

**SOA** Service-Oriented Architecture. 12, 13, 16

**URL** Uniform Resource Locator. 38, 39, 51, 57

**XML** Extensible Markup Language. 33