VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Mykola Zhabko

TRANSITIONING LEGACY DESKTOP SOFTWARE TO A WEB-BASED APPLICATION WITH ENHANCED INTERFACING

Technology and Communication
2024

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Mykola Zhabko
Title	Transitioning Legacy Desktop Software to a Web-Based Application with Enhanced Interfacing
Year	2024
Language	English
Pages	57
Name of Supervisor	Anna-Kaisa Saari

This thesis delves into the conversion process of transitioning outdated desktop software into a contemporary web application. The objective is to modernize the user interface and broaden its interfacing capabilities with other applications and databases. The core focus is investigating the obstacles, approaches, and advantages of migrating a software system from a conventional desktop setting to a more accessible and interconnected web-based platform.

The project focuses on migrating a Windows Desktop application, initially developed in C++ with the wxWidgets open-source framework for UI components. The existing application integrates with three external systems through SOAP clients, implemented as C++ libraries. Additionally, it relies on MS Access tables for database functionality.

The proposed transformation involves rewriting the application using Java 17 and Spring Boot 3.1.1, leveraging some existing SOAP web services, and introducing new functionalities through REST services. The revamped application is designed as a microservice and will be deployed within a Docker container. The development and maintenance processes are streamlined by adopting automated deployment practices, ensuring efficient releases, error reduction, and a seamless deployment experience for development and future maintenance tasks. The new application is set to utilize an Oracle database, with enhanced security measures implemented through the integration of OAuth 2.0.

The project aims to deliver an upgraded application with the same functionalities as legacy software, improved features, and a simplified structure for easy and cost-effective maintenance and future enhancements, facilitating seamless integration with external systems.

Keywords Legacy application migration, C++ to Java transition, microservice architecture, cost-effective solution

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

GAO	Government Accountability Office
FY	Fiscal Year
DME	Development, Modernization, and Enhancement
O&M	Operations and Maintenance
JPA	Java Persistence API
ORM	Object-Relational Mapping
SQL	Structured Query Language
CRUD	Create, Read, Update, Delete
JPQL	Java Persistence Query Language
HQL	Hibernate Query Language
SPA	Single Page Application
SOAP	Simple Object Access Protocol
REST	Representational State Transfer

1 INTRODUCTION

"Legacy" generally refers to something handed down from the past, often from generation to generation. Within the realm of software, this term typically denotes elements of the system having an old technology stack used for development and maintenance. Various components may be involved in such a system, including outdated applications, systems, code structures, or technologies from previous iterations.

This project involves upgrading legacy software from a desktop to a web-based application. The software, initially created by the customer of Wapice Ltd. in 2009, and which Wapice Ltd is currently maintaining, has become challenging to update and maintain, hindering the addition of new features and integration with other systems. Specific technologies will be chosen for this makeover to ensure a successful transition. The result will be a new web-based application with an automated deployment pipeline that streamlines maintenance processes, making it more cost-efficient. The new application will also be crafted for seamless integration with external systems, making it more versatile, and will include security measures such as multifactor authentication.

Businesses that continue to use outdated software may face failure due to delayed migration. This can result in significant costs if upgrades are not made in time. Procrastination can be detrimental to the growth and success of a company. Therefore, timely modernization efforts are essential to remain competitive in the market.

1.1 Research questions

This chapter outlines the research questions that guide the exploration of the transition from legacy desktop software to a modern web-based application at Wapice Ltd. The objective is to understand and address the technical, strategic, and operational challenges of such migrations. Each question aims to study a core aspect

of the migration process, providing a foundation for the detailed analysis in subsequent chapters. The research questions are as follows:

Analyzing the legacy application:

Question: What methodologies can be employed to analyze the codebase, integrations, and user interface of the legacy application to plan an effective migration strategy?

Purpose: To identify the best approaches for effectively dissecting the existing system to inform migration planning.

Data migration best practices:

Question: What are the best practices for migrating data from a desktop-based database (like MS Access) to a centralized web-based database (like Oracle or MySQL)?

Purpose: To ensure data migration is handled securely, accurately, and without loss, maintaining data integrity in the new system.

Technology stack selection:

Question: How do we select the appropriate technology stack for the new web-based application considering the existing legacy system's capabilities and future scalability needs?

Purpose: To determine the most suitable technology stack that balances modern needs with legacy system compatibility.

Testing the migrated application:

Question: What testing strategies should be employed to ensure that the migrated application meets the performance and functionality standards of the legacy application?

Purpose: To validate the new system using JPA repository testing, integration testing, and consistency validation testing, ensuring that the new application replicates the trusted functionalities of the legacy system and operates correctly within the new environment.

2 BENEFITS AND CHALLENGES OF MODERNIZING LEGACY SYSTEM

A legacy application or system is an information system that may be based on outdated technologies but is critical to day-to-day operations. Replacing legacy applications and systems with systems based on new and different technologies is one of the information systems professional's most significant challenges. As enterprises upgrade or change their technologies, they must ensure compatibility with old systems and data formats that are still in use. /1/

Legacy systems are often labeled as "outdated" or "obsolete," yet these terms do not fully capture their complexities. The consideration of legacy systems involves evaluating their performance, adherence to modern business protocols, compatibility, security, and the financial implications of their upkeep. Deciding when to upgrade such systems demands an in-depth analysis of these factors to ensure they meet the current and future requirements of business operations and technology trends./2/

2.1 Expenses Associated with Outdated Software

According to the United States Government Accountability Office (GAO), the federal government annually invests over $100 billion in IT and cybersecurity, with around 80% dedicated to operational and maintenance expenses of current IT infrastructures, as seen in **Figure 1** outdated legacy systems are also part of the spending pattern. The spending pattern indicates the pressing requirement for a strategic shift towards modernizing IT systems./3/

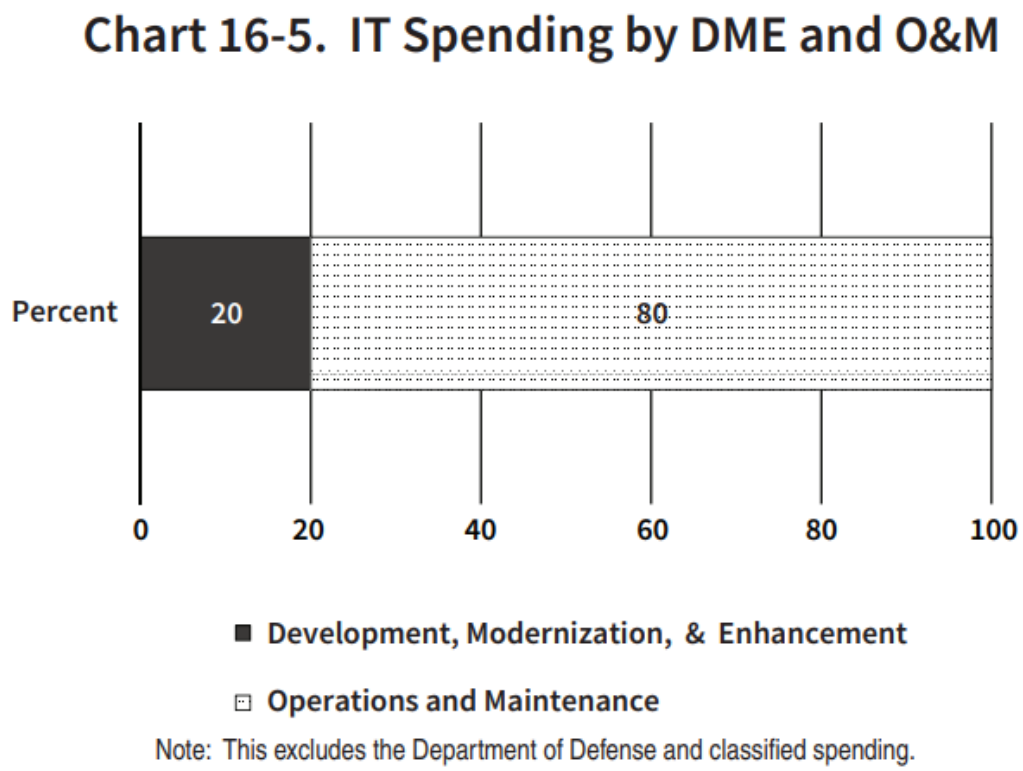

Figure 1. Distribution of Government-Wide IT Funding Between DME and O&M for FY 2019

Upgrading outdated software is crucial despite its high cost and time requirement, as relying on legacy systems for essential business operations can be risky and expensive. To determine if modernization is necessary, current software limits should be evaluated against business's efficiency and potential growth. /8/

2.1.1 Maintenance Cost

Legacy systems are complex to update and modify due to their monolithic architecture, which does not easily allow for changing or replacing individual components. Even minor updates can result in widespread system conflicts, rendering time-consuming and expensive alterations. Additionally, supporting and maintaining these systems, which rely on outdated technologies, requires specialized skills. As the original developers retire or transition to newer technologies, recruiting

and retaining skilled personnel becomes increasingly challenging, often leading to substantial costs associated with staff training. /4/

2.1.2 Integrating Legacy Software

Nowadays, modern technologies are designed to be easily integrated with external services, using APIs that provide immediate support for a wide range of programming languages and development frameworks. However, legacy software, built using outdated or uncommon technologies, presents significant limitations regarding compatibility and integration with third-party services. The rigid architecture of old systems makes it difficult to directly integrate with modern APIs, which is crucial in today's software development. Combining old software with newer tools requires custom coding, which is laborious, costly, and uncertain. Even with significant effort, success in achieving integration cannot be guaranteed. /5/

2.1.3 Security

Legacy IT infrastructure is a big security concern for many businesses and government areas because hackers often target them. These systems are essential for daily operations, making it complicated and expensive to replace or shut them down. Security experts must cooperate with companies to determine which systems are most important to fix or secure. Sometimes, if a system cannot be updated, it must be isolated to keep data safe and to watch more closely for any security threats. /6/

2.2 Defining Legacy System: Core Attributes and Significance

This chapter explores the process of identifying legacy systems accurately. It delves into the defining characteristics and challenges associated with these systems and provides practical tips for categorizing them to help organizations manage, modernize, or replace them effectively.

2.2.1 Obsolete Technologies

Obsolete software development technologies refer to the tools, languages, and methods that were once innovative and cutting-edge but need to be updated due to newer and better options. Although these outdated technologies played a significant role in creating many essential software programs in the past, they are now rarely used for new projects. For instance, programming languages like COBOL and Fortran were groundbreaking and crucial for early computer software, but they are no longer in demand today. Similarly, some old frameworks and databases need to catch up with today's needs for security and the ability to work on the internet flexibly, making them outdated as well. These outdated technologies are often found in legacy applications - software still in use despite being built on outdated tech. Such software can be legacy if it depends on old technology that is no longer supported, needs help to work with newer systems or requires specialized knowledge that is hard to find. Despite being operational, legacy software can be costly to maintain, perform poorly, and pose security risks. /7/

2.2.2 Poor Architectural Design

Legacy applications with poor design can lead to issues such as slow performance, higher chances of security risks, difficulties working with newer technologies, and adding new features. These problems usually arise from outdated methods, designs that do not allow easy modifications, too many fixed connections between parts, or inadequate instructions on how the system operates. /8/

2.2.3 Limited Interface Opportunities

Restricted interfacing in old systems is a problem where help is needed to connect or communicate effectively with third-party applications, software, or modern technologies. This limitation often arises because legacy systems were designed with older standards that are no longer widely used or supported. Integrating these applications with new systems, databases, or services becomes challenging.

This can lead to the isolation of legacy systems, making data exchange cumbersome and hindering the adoption of new technologies or processes that could improve efficiency and productivity. /8/

2.2.4 Code Degradation

Code degradation in legacy systems refers to the gradual decline in the quality and performance of the software code over time. This occurs as the system undergoes updates, modifications, or patches without a comprehensive understanding of the original design or consideration for long-term maintainability. Over time, these changes can lead to increased complexity, reduced readability, and the introduction of new bugs or vulnerabilities. As a result, the system becomes more challenging to maintain, update, or integrate with modern technologies, leading to decreased efficiency and increased costs for organizations relying on these legacy systems. /8/

2.2.5 Unfriendly User Interface/Experience

Legacy systems often have outdated user interfaces that lack the responsiveness, simplicity, and visual appeal of the contemporary application. This can result in a steep learning curve, decreased productivity, and user frustration, as the interfaces do not meet current expectations for usability and accessibility. The disparity between the user experience offered by modern applications and legacy systems can significantly impact user satisfaction and efficiency. /9/

2.2.6 Weak Performance

Weak performance in legacy systems poses significant issues, primarily due to outdated hardware compatibility, inefficient code, and architectures that cannot leverage modern processing power or technologies. These systems often have slow response times, the inability to handle concurrent users efficiently, and difficulties processing large volumes of data. As a result, organizations face operational inef-

ficiencies, reduced productivity, and a failure to meet the demands of current business environments. This performance lag hampers user satisfaction and limits the system scalability and adaptability to new requirements, making it a critical concern in maintaining and upgrading legacy systems. /10/

3 BACKGROUND

This chapter explores the genesis and objectives of the customer's original systems, the technological landscape, and the subsequent developments that necessitated an upgrade. It highlights the imperative for continuous innovation and adaptation to remain competitive and efficient and sets the context for the customer's strategic shift to a modern web-based application framework.

3.1 Introduction to the Legacy Application

In 2009, the customer developed an application called Trinity, which their in-house development team maintained. Later, the responsibility of maintaining and updating Trinity was transferred to Wapice Ltd, which now oversees its operations.

The application aims to receive data from three external systems, parse, and transform it into other data on which engineers can operate further via the app's UI. This data is related to customer products, and by utilizing Trinity, the responsible engineer can search existing product units or create new ones and update the data about product units to three external systems. Trinity's application name came from the fact that it is synchronizing three external systems that hold information about the product.

3.2 Current Architecture and Technologies

The aimed application name is Trinity. Its architecture consists of a Presentation layer (GUI) implemented using the wxWidgets framework, an application logic layer written in C++, which contains the application's core functionality, and a data access layer utilizing Open Database Connectivity to interact with the database. The database platform is MS Access Tables. **Figure 2** shows how Trinity integrates with external web systems using custom C++ libraries for SOAP web services and desktop Master application, triggering Trinity to start.

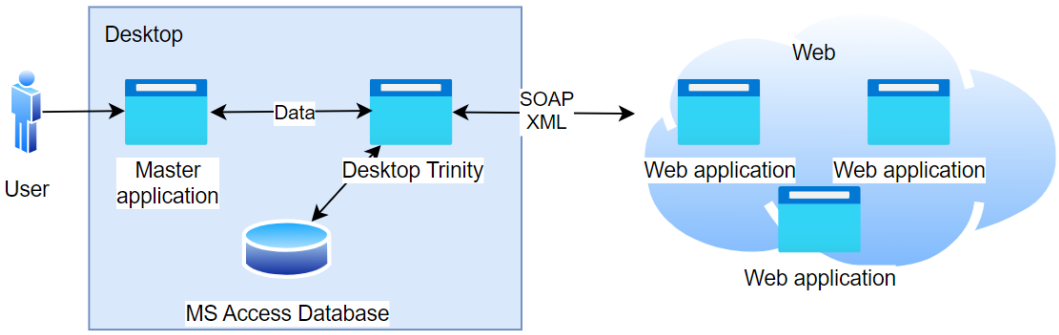

Figure 2. Trinity Integration Diagram.

3.3 Limitations of Trinity Application

This chapter will discuss why the customer wants to migrate Trinity to a web platform. The customer's business processes are evolving and require integrating modern technologies into their systems. However, Trinity, a core desktop application developed in C++, is increasingly becoming a bottleneck due to its limited integration capabilities. Trinity can only directly interact with other systems with complex and often inefficient workarounds, such as developing custom C++ libraries for each new integration. Transitioning to a web-based platform will address these integration challenges head-on, making the system more adaptable and ready for future technological improvements. Moreover, a new application can include all the required features missing from desktop Trinity, and other problems with maintenance or selecting a proper database solution can be solved in the same migration process.

3.3.1 Integration Needs

The customer plans to replace the Master application shown in **Figure 2** with the web application Loupe shown in **Figure 3**. This decision is part of the customer's broader strategy to shift towards web-based technologies across their business operations. Maintaining Trinity's current desktop format presents significant integration challenges and limitations considering this shift. Updating the legacy system would require complex and costly temporary fixes that only provide short-term solutions. These workarounds would accumulate and increase the system's

complexity, making developing and integrating new features cumbersome. Therefore, re-platforming Trinity to a web-based application is crucial for seamless integration with Loupe and aligns with the customer's strategic move towards a more modern, web-centric infrastructure. This transition will enhance the scalability and maintainability of the system, supporting the customer's long-term business objectives. **Figure 3** illustrates the integration of Web Trinity with Loupe and other external systems. This diagram visually represents the interconnected environment that the migrated Web Trinity will operate within.

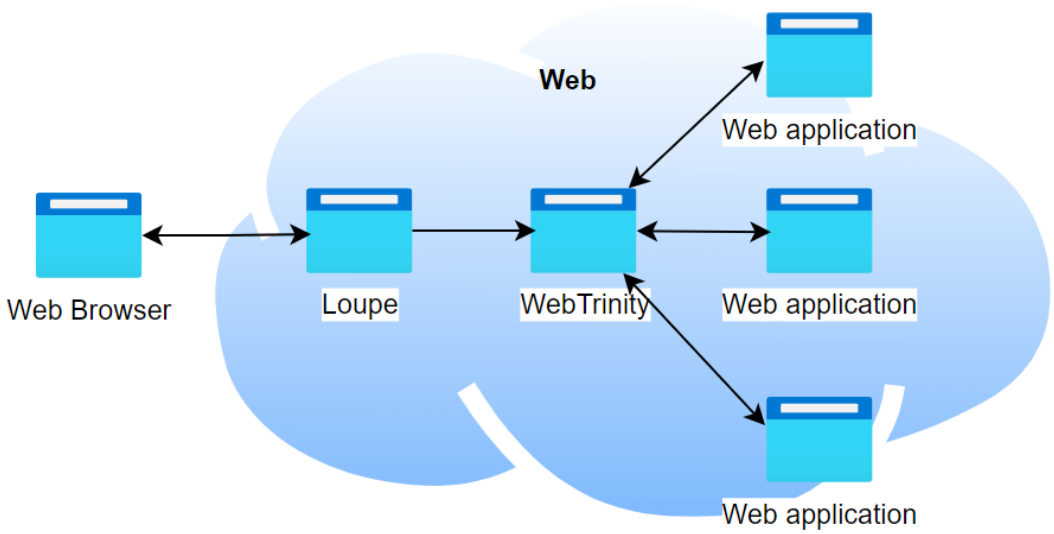

Figure 3. Web Trinity Integration Diagram

The customer will continue using the desktop version of Trinity while the Web Trinity is under development. The web version will be deployed alongside the desktop version for extensive testing. This thesis will concentrate only on migrating Trinity to the web platform and will not address the integration of Loupe with Web Trinity. The Master application will act as the central component that initiates Web Trinity.

3.3.2 Database Limitations

The legacy application uses MS Access tables as a database. However, this approach poses several challenges that affect operational efficiency and security.

One primary challenge is a cumbersome process related to updating data within the system. The process of updating data is shown in **Figure 4**, where a super user who can modify a database file has to first send the modification to the Wapice developer, who will add changes to version control, commit it, and send it back to the super user, super user further testing if the modifications applied correctly and send the tested file to inner customer team for publishing an updating to the server so the regular user can pull an updated file. A regular user is not allowed to modify a database. Currently, updates require the involvement of two external teams, which significantly complicates a simple task. As a result, the update process is time-consuming and error-prone, negatively impacting the application's reliability and the accuracy of the data it manages.

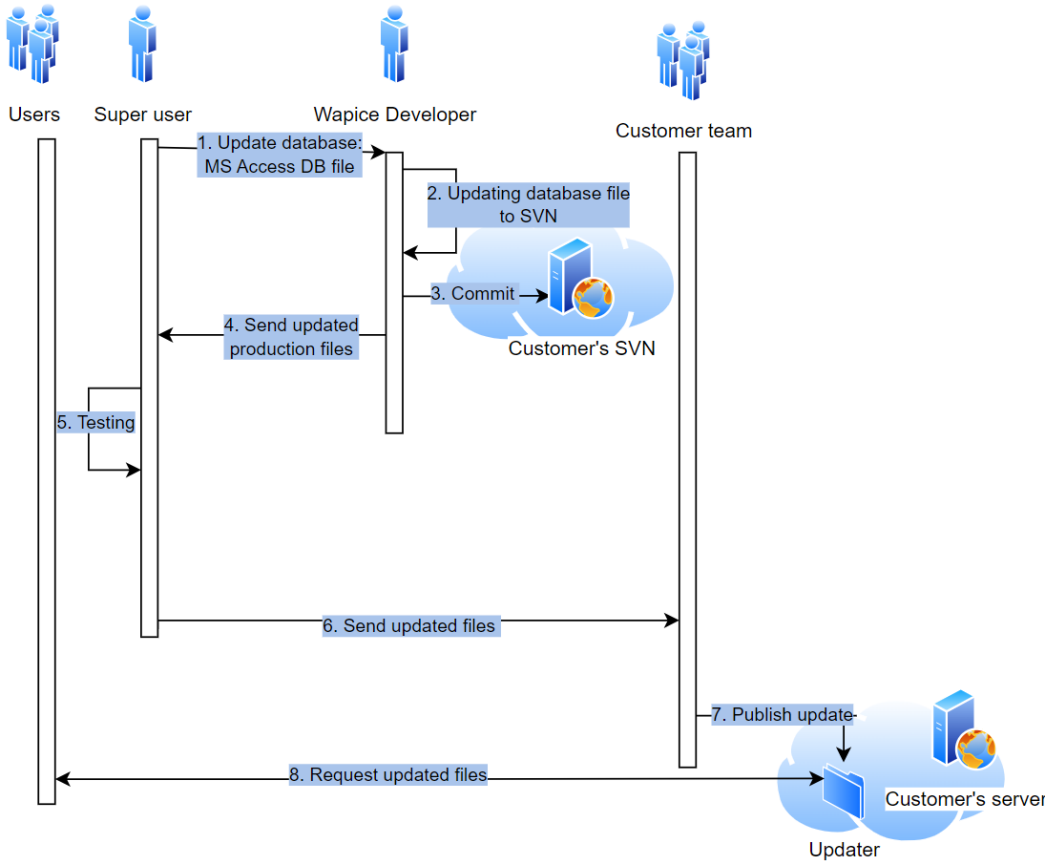

Figure 4. Process of Updating Database Tables

Moreover, the use of MS Access tables introduces a significant security risk. The lack of robust access controls puts the database at risk of unauthorized access and

modification. This practice does not align with best data security and integrity practices, posing a risk to the processes that rely on data from the legacy application. Unauthorized modifications could lead to incorrect data being provided into critical business processes, resulting in operational disruptions and potential financial losses. Therefore, it is essential to find a more secure, efficient, and modern database solution that can support the legacy application's requirements without compromising data integrity or security.

3.3.3 The High Costs of Maintenance and Upgrades

Maintaining a legacy desktop application developed in 2009 using C++ and wxWidgets 2.8.12 is a challenging task that requires technical acumen and an understanding of legacy technology. The development environment setup is complex, and it can be a barrier to entry for new developers. Currently, the maintenance of the application is the responsibility of a single developer, which amplifies the risks associated with knowledge silos and personnel dependency. Migrating the application to a more contemporary technology stack can make it more accessible to a broader array of developers and reduce the overhead costs associated with its upkeep. This transition is a strategic necessity that can redefine the role of the appalication in the organization, ensuring its relevance and utility in the face of evolving business needs and technological landscapes.

4 REQUIREMENTS

This chapter focuses on the specific requirements provided by the customer, crucial for the application's evolution. Although the core functionality remains intact, adding new features is essential to meet these requirements. These enhancements are not merely updates, but are crucial for unlocking new business opportunities and increasing the application's flexibility within the customer's operational environment.

A structured summary of the requirements is provided below, emphasizing the balance between preserving the application's essence and integrating new capabilities.

4.1 Preservation of Original Logic

The objective here is to maintain the integrity and functionality of the original application logic throughout the migration process. This includes all computational algorithms, data processing methods, and business rules that constitute the core operational logic of the existing system.

This is done because the original application logic represents the organization's accumulated domain knowledge and operational expertise. Preserving this logic is crucial for ensuring the migrated application meets the operational requirements and service levels expected by users and stakeholders.

The implementation is by conducting a thorough analysis of the existing application.

4.2 Data Exchange and Interaction with External Desktop Applications

The objective here is to ensure the migrated web-based application can open in conjunction with other desktop applications, receiving a dataset for processing, allowing users to perform selections through its user interface, and subsequently passing the results back to the initiating application.

The capability to interact with external desktop applications is a critical feature that supports integrated workflows and data exchange between different software tools used within the organization.

The implementation is by developing an interface or API that facilitates seamless data exchange between web-based and desktop applications. This might involve creating web services or middleware to communicate with desktop applications, ensuring compatibility and efficient data transfer.

4.3 Development of a Single Page Application (SPA)

The objective here is to design and implement the user interface (UI) of the new web-based application as a Single Page Application (SPA) that operates within a web browser. This approach aims to replace the traditional C++ desktop application UI, offering a modern, responsive user experience.

This is done because transitioning to a SPA format aligns with contemporary web development practices, enhancing user interaction and application performance. SPAs allow for dynamic updates to the UI without requiring page reloads, providing a seamless and intuitive experience like desktop applications.

The implementation is by selecting a modern front-end framework (React, Angular, Vue.js) based on community support, development efficiency, and stack compatibility. Design a user-centric UI that is intuitive, responsive, and accessible, retaining critical workflows from the legacy system for user familiarity while integrating enhancements. Develop or utilize APIs for backend integration.

4.4 Database Migration to Relational Database

The objective here is to transition the application's data storage from Microsoft Access tables to a relational database system hosted on the customer's server. This includes performing a comprehensive data migration to ensure all existing data is accurately transferred and integrated into the new database environment.

This is done because the move to a relational database is driven by the need for a more robust, scalable, and secure data management solution. Relational databases offer advanced features that support complex data structures, high-volume transactions, and enhanced data integrity. This upgrade addresses the limitations of MS Access, particularly in terms of scalability, security, and performance, making it a strategic choice for supporting the application's evolving data needs.

The implementation is by conducting a comprehensive analysis of the current data architecture in MS Access, examining tables, relationships, and data types. Formulate a detailed migration plan to outline the necessary steps for transferring data to a relational database, prioritizing compatibility and minimal operational disruption. For data migration, employ tools and scripts to move data from MS Access to the new system.

4.5 Direct Data Management Interface

The objective here is to implement an interface within the existing SPA to enable authorized users to directly enter and remove database data, bypassing the slower, manual update process via MS Access tables.

This is done because transitioning to a direct data entry interface significantly speeds up the data update process, eliminates the need for distributing files through third parties, and enhances data integrity by reducing manual handling errors.

The implementation is by developing a user-friendly interface integrated with the Single Page Application, ensuring only secure access for authorized users. Utilize web technologies for a seamless experience, implement strict input validation to maintain data integrity, and roll out features.

4.6 Authentication and Authorization

The objective here is to implement authentication and authorization for the new web-based application using the customer's existing internal application designed for authentication purposes. This integration ensures that access to the application is securely managed and that users have appropriate permissions depending on their roles.

This is done because leveraging the customer's existing infrastructure for authentication and authorization ensures a unified security approach across the organization. It simplifies user management, maintains consistency in security practices, and leverages established protocols. This strategy is crucial for protecting sensitive data and functionality within the application, providing a reliable means of verifying user identities and managing access rights.

The implementation is by developing an interface or service to integrate the web-based application with the customer's internal authentication system, possibly through API integration or shared security libraries. Implement a login mechanism within the web-based application that leverages the internal system for user authentication, including verifying user credentials and managing session states to secure access. Utilize the internal application to define and manage user roles and permissions, incorporating role-based authorization checks within the web-based application to restrict access to features and data according to user permissions.

5 TECHNOLOGIES

Selecting the right technology stack for migrating a legacy desktop application to a web-based platform involves carefully considering team expertise, system scalability, and integration capabilities. Chosen stack for this project includes Java with Spring Boot, Spring Data JPA, Angular, Oracle Database, Docker, and Rancher. It aligns with the project's needs for several vital reasons.

Java and Spring Boot provide a robust backend foundation, streamlining complex system integrations and application configurations. Oracle Database was selected for its superior performance, reliability, and security features, essential for managing complex and voluminous data from the legacy system. Spring Data JPA offers seamless integration with Oracle, simplifying data operations and reducing boilerplate code. Angular was used for the front end to enhance user interaction through its dynamic and scalable framework, while Docker and Rancher handle application deployment and orchestration, ensuring consistent environments across all stages of development.

The comparative analysis presented in the metrics table shown in **Table 1** further substantiates the decision to select Spring Boot with Angular. Spring Boot with Angular aligns with the existing Oracle database and leverages Java's robustness, a cornerstone of enterprise-level applications. This stack compatibility with complex transactional systems and integration capabilities with established enterprise services prevents potential disruption and maximizes continuity in development practices. The metrics table highlights Spring Boot with Angular's high scalability and performance, making it an ideal choice for large-scale applications. The mature ecosystem of the framework, characterized by extensive community support and comprehensive tools, ensures that new developments and maintenance are streamlined and efficient.

Moreover, the development team's expertise in Java and familiarity with Angular's TypeScript-based framework can significantly reduce the implementation timeline and enhance the quality of the output.

While alternatives like the MEAN stack /20/ (MongoDB, Express.js, Angular, Node.js) and .NET Core /21/ with React provide viable options, they were deemed less suitable primarily due to the additional challenges they present in integrating with the existing robust Oracle database system and the learning curve associated with adopting new backend technologies. For instance, the MEAN stack would introduce complexities in handling straightforward transactions with Oracle Database using Java and Spring. Similarly, adopting .NET Core, although powerful, would necessitate a shift from the well-established Java ecosystem, potentially complicating the integration with existing enterprise services.

Technology stack illustrated on **Figure 5**.

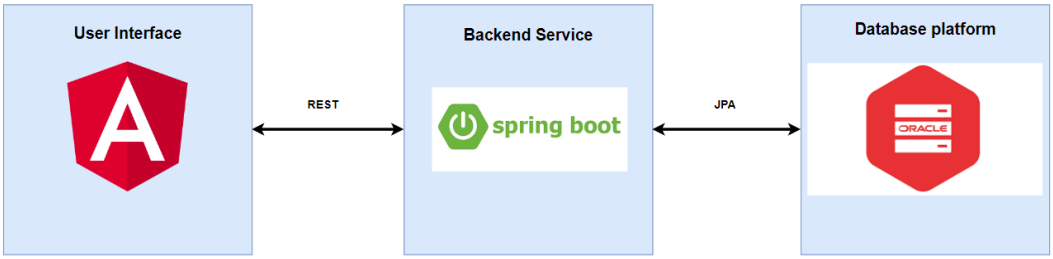

Figure 5. Technology stack

Table 1. Technology stack analysis

Metric	MEAN Stack	.NET Core with React	Spring Boot with Angular
Performance	Good performance due to Node.js non-blocking I/O model. MongoDB provides high read/write speeds.	High performance, especially on Windows servers. ASP.NET Core is optimized for low memory usage and high throughput.	Good performance, especially in large-scale applications due to Spring's robustness and Angular's efficient change detection mechanisms.
Scalability	Highly scalable with Node.js and MongoDB's horizontal scaling capabilities.	Scalable; .NET Core supports high-density scaling on cloud platforms.	Highly scalable; Angular supports lazy loading and Spring Boot makes it easy to scale microservices.

Learning Curve	Moderate; JavaScript-centric stack makes it accessible but full-stack JS can be complex for beginners.	Steeper for newcomers, especially those unfamiliar with C# and React's learning curve.	Steeper due to the complexity of both Spring Boot and Angular frameworks.
Ecosystem	Rich ecosystem with a vast number of libraries and tools in the JavaScript community.	Strong and well-supported, with extensive libraries and community support for both .NET and React.	Very mature ecosystem with comprehensive tools and community support for both Spring and Angular.
Development Cost	Generally lower due to open-source technologies and the prevalence of JavaScript developers.	Can be moderate to high depending on licensing costs for Microsoft technologies and developer salaries.	Moderate to high, considering the need for developers skilled in both Java and Angular.
Maintenance Cost	Moderate; JavaScript tools and libraries may change frequently, requiring regular updates.	Moderate; regular updates needed for .NET Core and React, but with stable support from Microsoft.	High; frequent updates in both Spring and Angular can lead to higher maintenance needs.
Industry Adoption	Widely adopted for small to medium web applications. Popular in startups and for quick MVP developments.	Strongly adopted in enterprise environments, especially for business applications requiring robust backend solutions.	Common in large enterprises and complex applications due to Java's stability and Angular's structured framework.
Development Team Expertise in current project	Moderate	Moderate	Hight

5.1 Java and Spring Boot

Java is an object-oriented programming language developed by Sun Microsystems in 1995. It is a platform-independent language with robust security features, comprehensive standard libraries, and automatic memory management. It is widely used in building enterprise applications, Android mobile applications, web and desktop applications, and embedded systems software. /11/

Spring Boot is an open-source framework that simplifies the creation and deployment of production-grade Spring applications. It offers built-in features for application development, including security, metrics, health checks, and externalized configuration. Spring Boot requires minimal upfront configuration and is widely used for fast application development in the Java ecosystem. /12/

5.2 Spring Data JPA and Hibernate

Spring Data JPA is a component of the Spring Data family, which aims to simplify data access in the Spring application framework. It uses JPA for ORM, making it easier for developers to implement repository interfaces, perform CRUD operations, and query databases without detailed SQL. Spring Data JPA abstracts the data layer, providing a more straightforward approach to data access and manipulation and automatically implementing common data access patterns through repository interfaces. Under the hood, it uses a JPA provider, such as Hibernate, to execute its operations. /13/

Hibernate is a Java ORM framework that maps Java classes to database tables, provides data query and retrieval facilities, and supports multiple ways of querying the database. It provides multiple ways of querying data, which include HQL, Criteria API, and Native SQL Queries. Along with these, JPA offers JPQL for executing database-independent queries. Spring Data JPA repositories simplify the process of writing code by offering various ways to define queries, such as method name conventions, annotations, and integration with Querydsl for intricate dynamic queries./14/

For this project, the development team used three querying methods from Spring Data JPA: the Method Name convention, the Native query, and the Entity Manager interface.

5.2.1 Method Name Convention with Spring Data JPA

Spring Data JPA simplifies data access by allowing developers to define repository interfaces using method signatures. Spring Data JPA automatically provides the underlying implementation of these methods and their database queries based on method names. Spring Data JPA's method naming convention is structured and intuitive. It uses keywords like find, count, and delete, followed by conditions that specify the query criteria. These conditions include clauses like By, GreaterThan,

LessThan, Between, Like, In, and more. For instance, a method named findUserByEmail in a UserRepository interface selects a user from the database whose email matches the provided argument. Spring Data JPA translates this method name into an SQL query using the JPA criteria API. /13/

5.2.2 Native Query

Java Persistence API provides an option to execute raw SQL directly against the database using native queries. This way, the abstraction layer provided by JPQL (Java Persistence Query Language) can be bypassed. Native queries help execute complex queries or leverage database-specific functions that JPQL does not support. The repository layer defines native queries using the @Query annotation with the nativeQuery = true attribute. Native queries allow developers to specify exact SQL statements for data operations, which offers precise control over database interactions. /13/

5.2.3 Entity Manager interface

The EntityManager interface in Java Persistence API serves as the primary point of interaction between a Java application and the database. It manages the lifecycle of entities, which are lightweight, persistent domain objects representing database rows. Through the EntityManager, developers can perform CRUD (Create, Read, Update, Delete) operations, query the database, and manage transactions. The interface provides methods for persisting entities to the database, merging changes, removing entities, and finding entities by their primary key. Additionally, the EntityManager supports JPQL (Java Persistence Query Language) for executing queries against entities. It is a factory for creating Query and TypedQuery instances, enabling dynamic and static query creation. /13/

5.3 Angular

Angular is a web application framework that is open-source and based on TypeScript. The Angular Team leads it at Google and a community of individuals and

corporations. The framework aims to provide a comprehensive solution for building dynamic single-page applications (SPAs), offering tools and libraries that cover everything from routing, forms, and client-server communication to testing utilities. One of Angular's strengths is its emphasis on code structure and modularity, which allows developers to create scalable and maintainable web applications. Additionally, Angular introduces powerful features like two-way data binding, dependency injection, and directives, making it a popular choice among developers for creating enterprise-grade web applications with complex functionality. /15/

5.4 Oracle Database

Oracle Database is a powerful and versatile database management system created by Oracle Corporation. It is known for its robustness, scalability, and ability to handle large volumes of data, making it an excellent choice for enterprise-level applications. The system is designed to support SQL for data manipulation and provides advanced features such as transaction control, data warehousing, and data encryption. Oracle Database can operate in cloud environments and on-premises, offering a flexible solution for data storage, analysis, and application development. The system is widely used across industries for critical business operations due to its reliability and extensive support for complex data management scenarios. /16/

5.5 Docker

Docker is an open-source platform that automates application deployment, scaling, and management within lightweight, portable containers. It simplifies managing applications across different systems and infrastructures, making it a popular choice for streamlining the development pipeline and implementing microservices architectures. /17/

5.6 Rancher

Rancher simplifies the deployment and operation of Kubernetes clusters by providing an open-source container management platform. It includes orchestration, networking, security, and storage to manage containers in a complete environment. Rancher offers a user-friendly interface to automate the deployment of Kubernetes clusters in different environments, such as on-premises, cloud, and hybrid setups. With its scaling, monitoring, and managing tools, Rancher enables organizations to run and manage containerized applications at scale efficiently. Its orchestration capabilities ensure that containers are automatically deployed and managed based on predefined policies and configurations, improving operational efficiency and resource optimization. /18/

6 SYSTEM DESIGN AND DEVELOPMENT PROCESS

To begin the migration process, it is crucial to understand the design of the system clearly. Often, applications interact with other applications or share data sources, necessitating a high-level analysis of the application environment to understand its context within the organization. This analysis involves gathering system information at a high level, identifying existing applications and their interactions, and evaluating their value. The next step is to perform microanalysis to examine application at the program level and understand program logic, business rules, code complexity, and data exchange schemas.

This analytical approach is chosen to ensure that the new system can effectively replicate and improve upon the functionalities of the legacy system. The approach provides both macro and micro perspectives of the system, which is necessary to avoid overlooking any critical component. While the high-level and microanalysis approaches formed the core of the migration strategy, several other methodologies, such as reverse engineering, dependency mapping, and prototyping, were considered but not chosen.

Reverse engineering involves deconstructing the application to discover its architectural and functional specifics without prior documentation. This method is beneficial when the source code and documentation are unavailable. However, reverse engineering is unnecessary in this case as there is access to documentation and Trinity's source code. Employing reverse engineering would have introduced unnecessary complexity and resource expenditure.

Dependency mapping involves using tools to visually map out all dependencies within an application, including libraries, external modules, and APIs, which is crucial for understanding how different parts of an application interact. Although dependency mapping is valuable, high-level analysis has already provided clear visibility into the system's integrations and dependencies. Additional mapping was considered redundant and not a productive use of limited resources.

Prototyping involves creating simplified versions of the new system early in development to test concepts and gather user feedback. Given the clarity of requirements and the depth of initial analyses, prototyping was not a priority. The focus was on a direct migration strategy with a clear roadmap, reducing the need for iterative concept testing typically associated with prototyping.

6.1 High Level Analysis

This chapter analyzes the legacy application at a high level, evaluating its structure, dependencies, and third-party integrations. The objective is to understand how the application starts and interacts with other applications or data sources. **Figure 6** shows how Trinity embeds with other systems. In the image, the Master application is a source of initial data, starting Trinity with that data in the payload. Then Trinity consumes three external web services using SOAP and REST APIs and has a database connection.

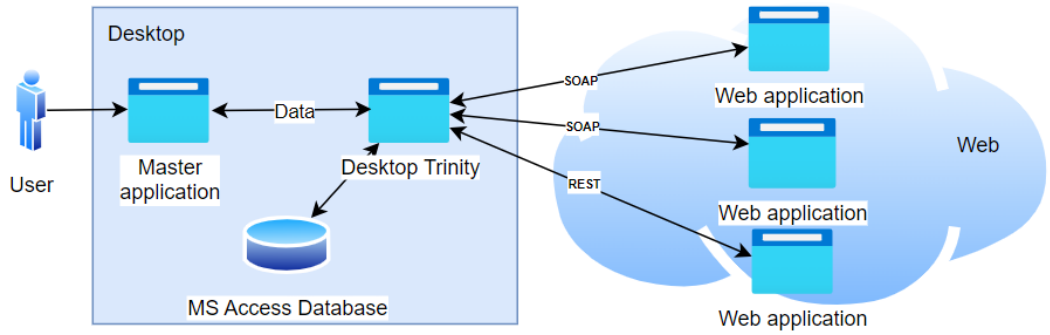

Figure 6. Trinity Interactions with other applications.

The target is to seamlessly replace Trinity application with a web version and conducted analysis shows that the web Trinity must have the same interactions with the three web applications, have a database connection, and be started by the Master application, as shown in **Figure 7**

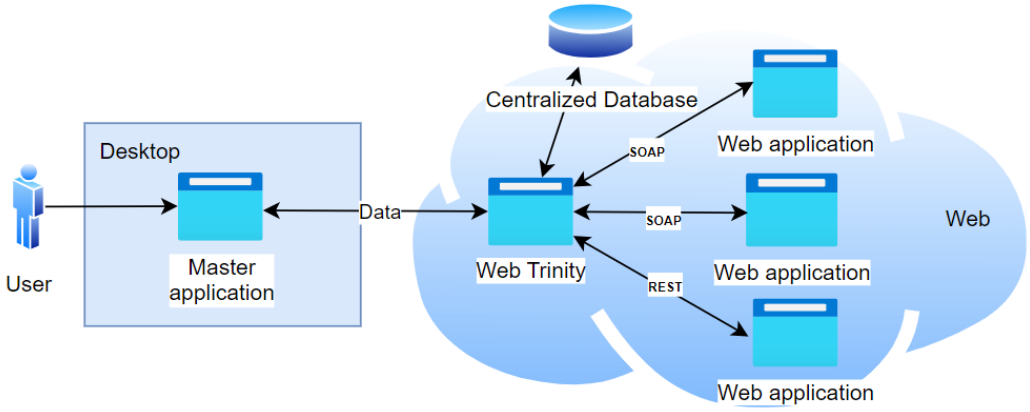

Figure 7. Web Trinity Interactions with other applications.

Due to re-platforming Trinity from desktop to web, features like handling user sessions, authentication, and authorization must be implemented also.

6.2 User Session

The Desktop Trinity application does not use a session-handling mechanism because the Master application handles it. When Trinity launches with initial data, it consumes REST and SOAP APIs provided by three external web systems and renders the user interface. The external web systems do not rely on the user's session. The user can open multiple instances of the Trinity app via the Master application, and they do not interact with each other, resulting in one open instance being a session. **Figure 8** depicts the described process, and the arrows pointing from the Master application to the opened Trinity application are sessions. The Master application is a source of incoming data to Trinity and expects a result from Trinity.

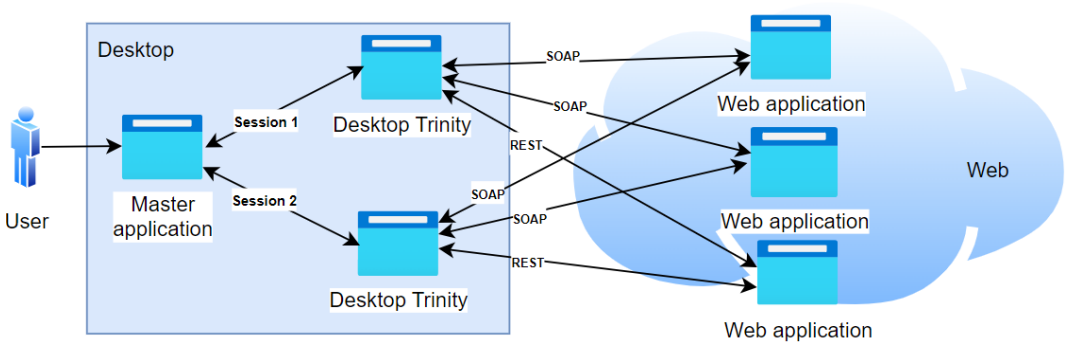

Figure 8. Session with Desktop Trinity Application

The customer's system necessitated additional features to handle user sessions when connected to Web Trinity. The Master application, which will utilize Web Trinity, was updated with a feature to establish a Web Socket connection and send data to Web Trinity, receiving a session ID and opening the browser with a session ID as a URL parameter. A web Socket connection is active until the user closes the browser tab associated with the session ID or the session timeouts after 30 minutes idle. After a timeout, the Web Trinity removes the data related to a session from its session storage.

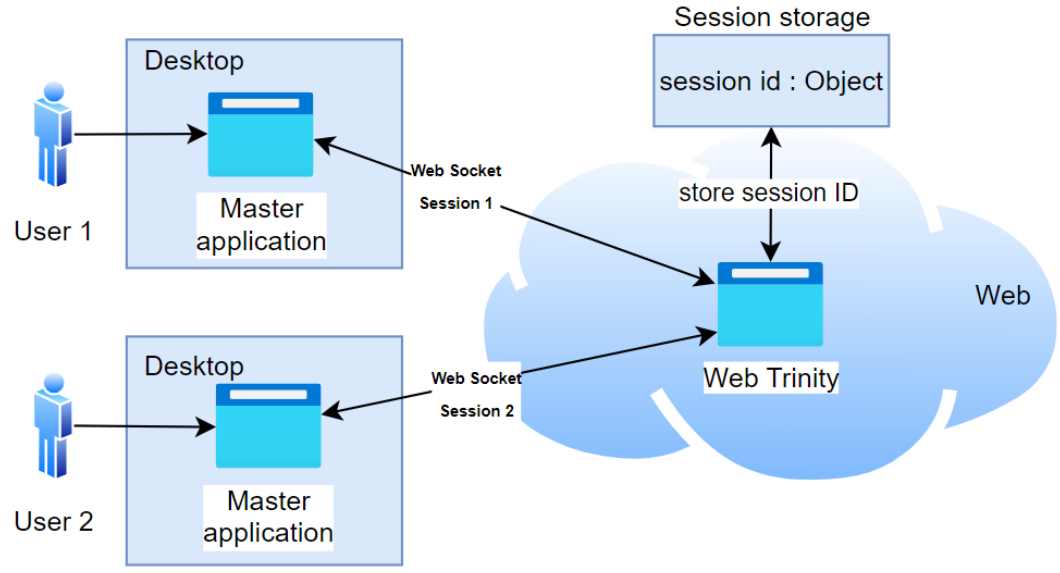

Figure 9. Session with Web Trinity Application

Figure 9 shows a simple high-level overview of the session handling design. Arrows pointing from the Master application are web socket connections that maintain a session for the user. The session storage is a map stored in the Web Trinity's RAM. The map contains key-value pairs where the key is a session ID and the value is an object created from the data sent by the Master application to Web Trinity.

To ensure secure and efficient communication between the user and Web Trinity, a session handling system needed to be designed. The process is shown in **Figure 10**.

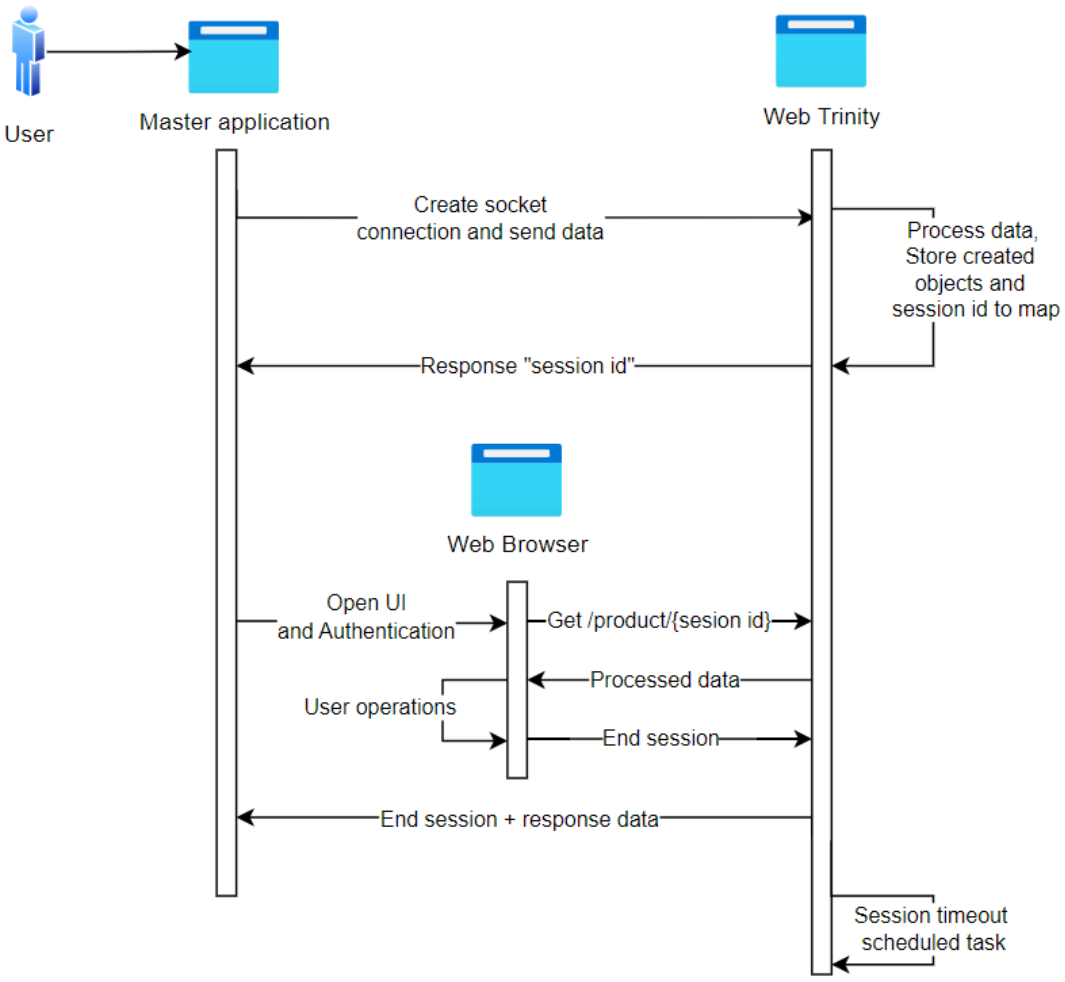

Figure 10.User Session Management in Web Trinity.

First, the User sends data to the Web Trinity via the Master application for processing. The server processes the data, creates the necessary objects, and gener-

ates a unique session ID. The application stores this session ID and the newly created objects in the server's memory as a map <session ID: object> and returns the session ID to the Master application. Next, when the Master application receives the session ID, it attempts to open the user interface with the session ID added as a URL parameter so that the server can retrieve the correct data corresponding to the session ID to build a UI. At this point, the User has to perform authentication. This advanced mechanism ensures that the server can present a personalized and secure user interface with data specific to the session ID. Once the User has completed their tasks and closes the browser tab, the data associated with the session ID is wiped out from the server memory to maintain privacy and security. The application also removes the session ID and related objects from the map after a 30-minute timeout or if the web socket connection drops from either side.

6.3 Analyzing Incoming Data and Code Base

The initial phase of the application involves processing incoming data, which is presented in a file format, as shown in **Figure 11**, containing key-value pairs. This format necessitates a parsing mechanism to interpret and utilize the data effectively within the application workflow. Next the application relies on three external interfaces. These integrations are crucial as they fetch additional data required for further parsing and processing. The interfaces include two that use SOAP, and one that utilizes REST principles. These integrations demonstrate the dependence of the application on external data sources to supplement its internal data processing and user interface construction.

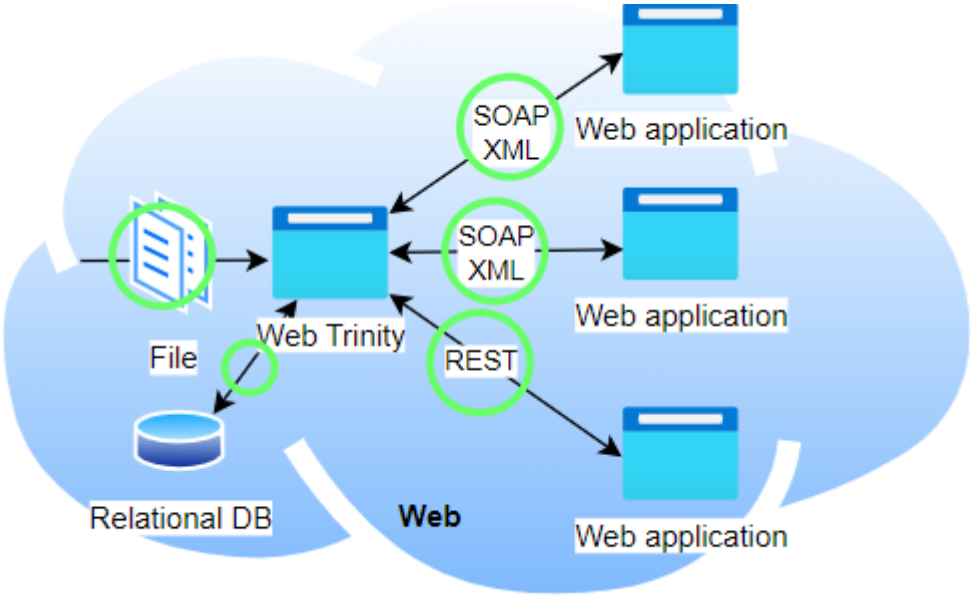

Figure 11. Integration Points for Reimplementation in Web Trinity.

As a result of the code analysis, all highlighted with green circles integration nodes shown on **Figure 11** must be designed and implemented for the new Web Trinity application.

6.4 User Interface and Data Transformation

The development team analyzed the user interface to determine which data is used to create it, which parts of the data can be modified by the user, what the user can add to it, and what user interactions are allowed with the data provided. This involved comprehensively examining user inputs, selections, and subsequent actions performed with the transformed data.

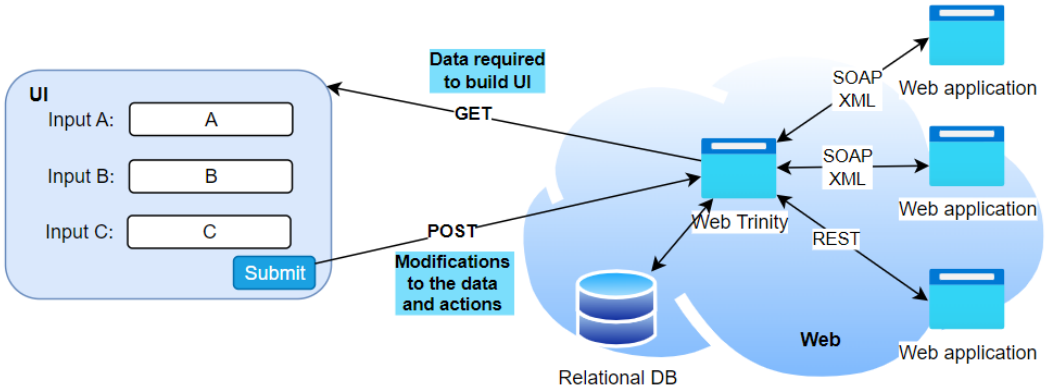

Figure 12. Client-Server Interaction for UI Rendering and Data Processing Workflow

Figure 12 illustrates the data required to build a UI as an arrow from Web Trinity to UI. This data is a composite object that includes lists, strings, Boolean values, and other objects. The user's actions and modifications are shown with an arrow pointing from UI to Web Trinity, meaning that all changes, selections, and requested actions go to the server for processing and delivery.

One of the primary objectives of this analysis was to identify the destinations for the transformed data to ensure that the UI supports the overall data processing and transfer objectives of the application. The study also aimed to identify any areas of the UI/UX that may require improvement.

Figure 13 shows an example of a possible addition to the UI. As an example from the project, some data was transferred to the UI and had to be shown in the form of a table, and in the legacy application, there were static placeholders for the table columns as the old UI framework did not support an easy way of rendering

dynamic data. With a modern UI framework, it was possible to render columns dynamically and improve the user experience.

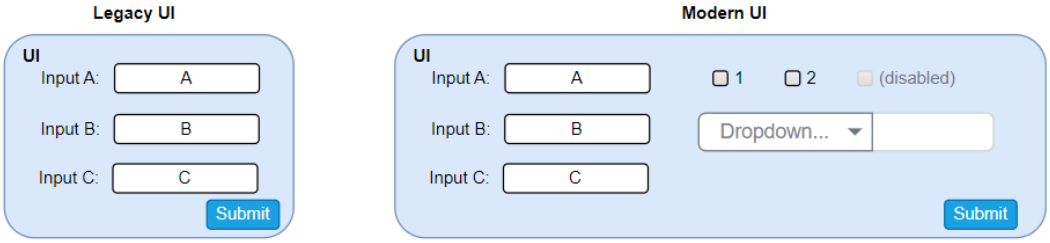

Figure 13. Example of the UI/UX improvements.

6.5 Database Interaction

A significant amount of data the application uses is stored in a database. The analysis has found that the existing SQL queries, initially designed to retrieve data from Microsoft Access tables, can be reused with minimal adjustments to query a relational database centralized on a server. This reuse of SQL queries allows for an efficient transition from a localized database structure to a more robust server-based relational database system.

To maintain consistency and dependability in data retrieval procedures, a strategy to retrieve data from the database in the same manner as it was executed in the previous application needs to be implemented. This will ensure that the application workflow stays unchanged.

6.6 From MS Access to a Centralized Relational Database

Migrating data from database like Microsoft Access to a centralized relational database requires careful planning and execution. In the initial migration phase, it is essential to evaluate the current structure of the database, its usage, and performance requirements. This assessment aids in selecting the right target database system and planning for necessary schema mapping and data type conversions. It is crucial to identify elements within the existing database that do not directly

translate to the web-based system, such as custom queries and reports, and plan modifications to these components accordingly.

Preparation for migration includes cleaning the existing data to ensure its accuracy, consistency, and completeness, which is vital for maintaining data integrity in the new system. Data is also backed up during this stage to prevent potential loss. The chosen method for data export, whether through built-in tools, custom scripts, or third-party solutions, depends on the complexity of the data and volume. The exported data might first be converted into an intermediate format like CSV or XML, which can then be transformed to meet the target database schema of the system and data type requirements.

The final stages of the database migration process involve importing the transformed data into the new database. It is crucial to ensure that the functionality and performance of the new database meet the planned specifications. Conducting trial imports will help fine-tune the import process and address any issues that may arise. Once the migration is complete, the database is thoroughly verified for accuracy and completeness, and necessary adjustments are made to optimize its performance. The new system is then precisely tested under operational conditions to ensure all functionalities perform as expected.

The development team used DBeaver for data migration from MS Access tables to Oracle Database in this project. DBeaver is a comprehensive database tool supporting multiple platforms, including MS Access and Oracle. /19/

The choice of DBeaver as the data migration tool was pivotal due to several factors that aligned with the project's requirements and goals. Primarily, the simplicity of the data involved in the migration significantly influenced this decision. The data set, devoid of complex constraints or relational dependencies, was ideally suited for a straightforward migration tool. DBeaver, known for its user-friendly interface and robust functionality across multiple database platforms, including MS Access and Oracle, provided an efficient pathway for handling such uncomplicated data

structures. Furthermore, DBeaver's capability to perform trial imports and facilitate easy schema comparison before the final migration added an essential layer of verification to the process.

The migration process began with establishing connections to both the source (MS Access) and the destination (Oracle DB). After securely setting up the connections, DBeaver's built-in export functions were used to automate the data transfer process. The DBeaver export wizard provided a straightforward, user-friendly interface to configure the migration details. Users could select specific tables or entire databases for migration. Advanced options allow for the customization of data types, handling of primary keys, and schema transformation as needed. The export functionality included robust error-handling capabilities to manage any issues encountered during the data transfer, such as data type mismatches or constraint violations.

6.6.1 Alternative Data Migration Options and Justification for Choosing DBeaver

Although DBeaver was used to migrate data from MS Access to Oracle DB for this project, other methods and tools are available. Few alternatives that could also be used and the reasons why the development team ultimately chose DBeaver are described below.

Oracle SQL Developer is a free, integrated development environment that provides tools for managing the Oracle database. It includes a feature to connect to MS Access and directly migrate tables to Oracle. While Oracle SQL Developer is a robust tool designed explicitly for Oracle databases, DBeaver offers a more generalized approach that is database agnostic, providing flexibility and support for multiple databases beyond Oracle, which could be helpful in multi-database environments.

MS Access provides options to export data to different formats, including Excel and ODBC databases, and directly to Oracle via ODBC connections. Exporting directly from MS Access can be less reliable for large or complex databases because it may require additional steps to configure ODBC connections properly and handle data type conversions manually. Therefore, it was decided not to be used.

Tools such as Informatica, Talend, or SSIS (SQL Server Integration Services) can be used to create more complex ETL pipelines that transfer data and transform it during the migration process. These tools are excellent for large-scale data warehousing projects but might need to be more efficient for straightforward migrations. They require more setup and are generally more resource intensive than DBeaver.

Writing custom scripts using languages like Python or Perl allows for highly customizable migration processes, where specific handling of data types and exceptions can be programmed. Custom scripting offers unlimited flexibility but requires significant programming expertise and time to develop and test. DBeaver's out-of-the-box functionality, minimal setup, and user-friendly interface provided a more efficient solution for this project's needs.

6.7 Spring Data JPA as Comprehensive Toolbox during Migration

Spring Data JPA played a vital role in the transition process by providing a repository abstraction that greatly simplified adapting and optimizing SQL queries for the new system. Spring Data JPA repositories were utilized to automate a significant portion of the data access layer. Almost half of the data retrieval operations were smoothly handled by adopting the repository's method naming conventions from Spring Data JPA, as shown in **Figure 14**, such as "findByFanm3". This naming convention approach to querying preserved the original SQL operations' intent, thus improving the code's clarity and maintainability.

Trinity (C++)

```
...


std::ostringstream query;
query.str("");
query<<"SELECT OuterDiameter FROM Fans ";
query<<"WHERE FAN_M3="<<fanIndex;

std::vector< std::vector<std::string> > results;

executeQuery(query, results);

....
```

Web Trinity (Java)

```
public interface FansRepository extends JpaRepository<Fans, Long>{

    Fans findByFanm3(Integer fanIndex);
}

...

Fans fan = fansRepository.findByFanm3(fanIndex);

....
```

Figure 14. Comparison between Legacy SQL Implementation in C++ Application and Modernization with Spring Data JPA Repository Method Naming Convention.

In situations where naming conventions were insufficient for about 40% of queries or needed additional customization, the "@Query" annotation was used, as seen in **Figure 15**. Annotation proved to be a flexible tool, it allowed to explicitly define the queries, which helped to ensure the accuracy of data retrieval mechanisms while also benefiting from the performance optimizations and integration offered by Spring Data JPA.

Trinity (C++)

```
....

std::ostringstream query;
query.str("");
query<<"SELECT Cable_set_ID ";
query<<"FROM cable_set ";
query<<"WHERE (P_MCODE LIKE '%,"<<p_mcode<<",%' OR UCASE(P_MCODE) LIKE '%,ALL,%') ";
query<<"AND (GEN_CODE LIKE '%,"<<gen_code<<",%' OR UCASE(GEN_CODE) LIKE '%,ALL,%') ";
query<<"AND DO1_min <= "<<do1<<" ";
query<<"AND (DO1_max >= "<<do1<<" OR DO1_max = 0) ";
query<<"AND (FRAME LIKE '%,"<<frame<<",%' OR UCASE(FRAME) LIKE '%,ALL,%') ";
query<<"AND (P_HXR LIKE '%,"<<adept.getDataAsInt("P_HXR")<<",%' OR UCASE(P_HXR) LIKE '%,ALL,%') ";
query<<"AND (Poles LIKE '%,"<<adept.getDataAsInt("2P")<<",%' OR UCASE(Poles) LIKE '%,ALL,%') ";
query<<"AND (InsulationClass LIKE '%,"<<insulClass<<",%' OR UCASE(InsulationClass) LIKE '%,ALL,%') ";
query<<"AND CableArea = "<<cableArea<<" ";

std::vector< std::vector<std::string> > results;

executeQuery(query, results);

....
```

Web Trinity (Java)

```
@Query(value = "SELECT * "
            + "FROM CABLE_SET "
            + "WHERE (P_MCODE LIKE ('%,' || :p_mcode || ',%') OR UPPER(P_MCODE) LIKE '%,ALL,%') "
            + "AND (GEN_CODE LIKE ('%,' || :gen_code || ',%') OR UPPER(GEN_CODE) LIKE '%,ALL,%') "
            + "AND DO1_min <= :do1 "
            + "AND (DO1_max >= :do1 OR DO1_max = 0) "
            + "AND (FRAME LIKE ('%,' || :frame || ',%') OR UPPER(FRAME) LIKE '%,ALL,%') "
            + "AND (P_HXR LIKE ('%,' || :p_hxr || ',%') OR UPPER(P_HXR) LIKE '%,ALL,%') "
            + "AND (POLES LIKE ('%,' || :poles || ',%') OR UPPER(POLES) LIKE '%,ALL,%') "
            + "AND (INSULATIONCLASS LIKE ('%,' || :insulationClass || ',%') OR UPPER(INSULATIONCLASS) LIKE '%,ALL,%') "
            + "AND CableArea = :cableArea", nativeQuery = true)
    List<CableSet> getCableSet(@Param("p_mcode") String p_mcode, @Param("gen_code") String gen_code, @Param("do1")double
do1,
            @Param("frame") String frame, @Param("p_hxr") String p_hxr,
            @Param("poles") String poles,@Param("insulationClass") String insulationClass,@Param("cableArea") String
cableArea);

.....

List<CableSet> cableSets = cableSetRepository.getCableSet(p_mcode, gen_code, do1, frame,
                params.valueAsString(AdeptAttributeNames.P_HXR), params.valueAsString(AdeptAttributeNames.POLES),
insulClass,
                cableParameters.get("cableArea"));
```

Figure 15. C++ SQL Queries compared with Spring Data JPA @Query Annotation with Parameterized Methods.

For the more complex cases, as shown in **Figure 16**, where dynamic queries were too difficult to handle via straightforward method naming or annotated queries, it was decided to utilize the Java Persistence API with the Entity Manager interface. This approach helped maintain the application's data access and performance while providing flexibility and type safety for intricate querying scenarios.

Trinity (C++)

```
....
    std::ostringstream virta;
    virta.str("");
    virta<<"SELECT CodeText,CodeNumber,P2,Q1,q,W,Kytk,e,Parallel,InternalConnection,";
    virta<<"Dahlander,Nlayer,Tapa,NVR_lkm,Kaaminta_tapa,Jaettu,Terminals,Konsentrinen, WMTYPE ";
    virta<<"FROM WindingDiagrams D WHERE ";
    if(poles.length()>0)
    {
      virta<<"D.P2='"<<poles;
    }
    else
    {
        virta<<"D.P2='"<<adept.getDataAsString("2P");
    }
    virta<<"' AND D.Q1="<<adept.getDataAsInt("Q1");
    if(parallel.length()>0)
    {
        virta<<" AND D.Parallel='"<<parallel<<"'";
    }
    else
    {
        virta<<" AND D.Parallel='"<<adept.getDataAsInt("A1")<<"'";
    }
    virta<<" AND D.Nlayer="<<adept.getDataAsInt("NLAYER1");
    if(connection.length()>0)
    {
        virta<<" AND D.Kytk='"<<connection<<"'";
    }
    else
    {
        virta<<" AND (D.InternalConnection=0 OR (D.InternalConnection=1 AND D.Kytk LIKE '%";
        switch(adept.getDataAsInt("CONNECT1"))
        {
        case CONNECT1_DELTA: case CONNECT1_DELTA_DAHLANDER:
            virta<<"D";
            break;
        case CONNECT1_STAR: case CONNECT1_STAR_DAHLANDER:
            virta<<"Y";
            break;
        }
        virta<<"%'))";
    }
    if(poles.length()>0)
    {
        virta<<" AND D.Dahlander=1";
    }

    if(adept.getDataAsInt("NLAYER1")==2)
    {
        virta<<" AND D.W='"<<adept.getDataAsInt("YPITCH1")<<"'";
    }
    if (isMachineWindingDiagrams)
    {
        virta<<" AND (D.WMTYPE IS NULL OR NOT D.WMTYPE='1')";
    }
    virta<<" AND (";
    virta<<" EXISTS (SELECT 1 from WindingDiagramsSite S";
    virta<<" WHERE D.CodeText = S.CodeText";
    virta<<" AND D.CodeNumber = S.CodeNumber";
    virta<<" AND S.Site = "<<adept.getDataAsInt("SITE")<<")";
    virta<<" OR";
    virta<<" NOT EXISTS (SELECT 1 FROM WindingDiagramsSite S";
    virta<<" WHERE D.CodeText = S.CodeText";
    virta<<" AND D.CodeNumber = S.CodeNumber)";
    virta<<")";

    virta<<" ORDER BY CodeNumber";
    std::vector< std::vector<std::string> > results;
```

Figure 16. Complex SQL query in legacy Trinity application.

Figure 17 shows how the approach of using Entity Manager directly preserved the original query's logic by dynamically constructing and executing the query based on the specified conditions, ensuring a seamless transition and system reliability.

Web Trinity (Java)

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.Query;

public class WindingDiagramsRepositoryImpl implements WindingDiagramsRepositoryExtension {
    @PersistenceContext
    private EntityManager em;
    @SuppressWarnings("unchecked")
    @Override
    public List<WindingDiagrams> searchWindingDiagrams(Params params, String poles, String parallel, String connection,
            boolean isMachineWindingDiagrams) {
        String virta = "";
        virta += "FROM " + WindingDiagrams.class.getName() + " D WHERE ";
        virta += "D.p2= :p2";
        virta += " AND D.q1= :q1";
        virta += " AND D.parallel= :parallel";
        virta += " AND D.nlayer= :nlayer";
        String connect1 = "";
        if(connection.length()>0)
        {
            virta += " AND D.kytk='" + connection + "'";
        }
        else
        {
            virta += " AND (D.internalconnection=0 OR (D.internalconnection=1 and D.kytk LIKE :connect1 ))";
            switch(params.valueAsInt(AdeptAttributeNames.CONNECT1))
            {
            case AdeptParameters.CONNECT1_DELTA: case AdeptParameters.CONNECT1_DELTA_DAHLANDER:
                connect1 = "%D%";
                break;
            case AdeptParameters.CONNECT1_STAR: case AdeptParameters.CONNECT1_STAR_DAHLANDER:
                connect1 = "%Y%";
                break;
            }
        }
        if(poles.length()>0)
        {
            virta += " AND D.dahlander=1";
        }
        if(params.valueAsInt(AdeptAttributeNames.NLAYER1)==2)
        {
            virta += " AND D.w= :ypitch1";
        }
        if (isMachineWindingDiagrams)
        {
            virta += " AND (D.wmtype IS NULL OR NOT D.wmtype='1')";
        }
        virta += " AND (";
        virta += " EXISTS (SELECT 1 from " + WindingDiagramsSite.class.getName() + " S";
        virta += " WHERE D.codetext = S.codetext";
        virta += " AND D.codenumber = S.codenumber";
        virta += " AND S.site = :site)";
        virta += " OR";
        virta += " NOT EXISTS (SELECT 1 FROM " + WindingDiagramsSite.class.getName() + " S";
        virta += " WHERE D.codetext = S.codetext";
        virta += " AND D.codenumber = S.codenumber)";
        virta += ")";
        virta += " ORDER BY codenumber";
        Query q = em.createQuery(virta);
        q.setParameter("connect1", connect1);
        q.setParameter("site", params.valueAsInt(AdeptAttributeNames.SITE));
        q.setParameter("nlayer", params.valueAsInt(AdeptAttributeNames.NLAYER1));
        q.setParameter("q1", params.valueAsInt(AdeptAttributeNames.Q1));

        if(poles.length()>0) {
            q.setParameter("p2", String.valueOf(poles));
        } else {
            q.setParameter("p2", String.valueOf(params.valueAsString(AdeptAttributeNames.POLES)));
        }
        if(parallel.length()>0)
        {
            q.setParameter("parallel", String.valueOf(parallel));
        }
        else
        {
            q.setParameter("parallel", String.valueOf(params.valueAsInt(AdeptAttributeNames.A1)));
        }

        if(params.valueAsInt(AdeptAttributeNames.NLAYER1)==2){
            q.setParameter("ypitch1", String.valueOf(params.valueAsInt(AdeptAttributeNames.YPITCH1)));
        }
        return q.getResultList();
    }
```

Figure 17. Complex SQL query in Web Trinity application using EntityManager interface from Spring JPA.

Spring Data JPA and the Hibernate framework were instrumental in successfully migrating the application. These technologies provided a comprehensive and flexible set of tools that helped efficiently reimplement legacy SQL queries within the context of a Java-based web application.

7 TESTING

This chapter delves into the comprehensive testing strategy to ensure a flawless transition from the legacy C++ application to the newly developed web-based system. Central to the testing methodology is Spring Boot JUnit 5, the latest iteration of Spring's robust testing framework designed to integrate seamlessly with the Spring ecosystem.

Spring Boot JUnit 5 was selected due to its advanced features for writing and executing tests, including support for Java's latest features and more flexible configurations for test cases. The integration of JUnit 5 with Spring Boot is particularly advantageous for loading the minimal Spring context necessary for tests, enhancing the speed and efficiency of the testing process. This framework supports various testing needs, from simple unit tests to complex integration tests, making it an ideal choice for the project.

Its compatibility with the chosen technology stack heavily influenced the decision to utilize Spring Boot JUnit 5. Given the application's reliance on Spring technologies, such as Spring Data JPA for database interactions and Spring's web modules for REST and SOAP web services, JUnit 5 integrates naturally. It facilitates using Spring's dependency injection to manage test instances and configurations effectively.

7.1 JPA Repository Testing

Spring Boot simplifies JPA repository testing with JUnit 5, allowing developers to focus on testing business logic rather than boilerplate setup. The framework provides helpful annotations and utilities that automate the configuration of an in-memory database, scan entity classes, and set up repository contexts.

JPA Repository Test

```
@DataJpaTest
public class UserRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository userRepository;

    @Test
    void whenFindByUsername_thenReturnUser() {

        User newUser = new User("Name", 25);
        entityManager.persist(newUser);
        entityManager.flush();

        User found = userRepository.findByUsername(newUser.getUsername());

        assertAll("User",
            () -> assertEquals(found.getUsername(), newUser.getUsername()),
            () -> assertEquals(found.getAge(), newUser.getAge())
        );
    }
}
```

Figure 18. JPA Repository test

Figure 18 shows an example of a JPA Repository test. Such tests are straightforward but helpful during the migration process when gradually implementing the logic from the legacy application, where each subsequent implementation step requires retrieving data from another database table. A developer can apply a Test-driven development approach to confirm that the developer correctly created the Entity classes and that repository interface methods can store and retrieve data from database tables.

7.2 Integration Testing

Integration testing is a crucial step in migration. It validates the ability of the new application to communicate with external APIs while maintaining data integrity. This subchapter explains how to conduct integration tests and exemplifies it with

a detailed test case. The testing confirms that the app meets functional requirements and exhibits resilience and accuracy in real-world scenarios.

Integration Test

```
@SpringBootTest(classes = { AdeptReaderWebserviceV4Application.class })
public class EaClientIntegrationTest {

    @Autowired
    private EaClient eaClient;

    @Test
    public void CalculationIdControlValue_EditSaveVerifyRevert_Test()
            throws IOException {

        List<ControlValue> controlValueFromEa = eaClient.getControlValue("LKH86756");

        ControlValues controlValues = ControlValues.ofControlValue(controlValueFromEa.iterator().next());

        assertNotNull(controlValueFromEa);
        assertEquals(controlValueFromEa.iterator().next().getCalculationNumber(), "LKH86756");
        assertEquals(controlValueFromEa.iterator().next().getFn(), 50);

        assertNotNull(controlValues);
        assertEquals(controlValues.getCalculationNumber(), "LKH86756");
        assertEquals(controlValues.getFn(), 50);
        assertEquals(controlValues.getR1v(), 12.178);

        assertEquals(controlValues.getR1v(), controlValueFromEa.iterator().next().getR1v());

        controlValues.setR1v(12.179);

        eaClient.storeControlValue(controlValues, "test1");

        List<ControlValue> controlValueFromEaAfterModification = eaClient.getControlValue("LKH86756");

        assertEquals(controlValueFromEaAfterModification.size(), 1);

        assertEquals(controlValueFromEaAfterModification.iterator().next().getR1v(), 12.179);

        controlValues.setR1v(12.178);

     eaClient.storeControlValue(controlValues, "test1");

        controlValueFromEaAfterModification = eaClient.getControlValue("LKH86756");

        assertEquals(controlValueFromEaAfterModification.iterator().next().getR1v(), 12.178);

    }
```

Figure 19. Integration Test

The integration test shown in **Figure 19** demonstrates external API interactions through the **EaClient** service. It retrieves a control value by its calculation number, modifies and saves it remotely, and reverts it to its original state. This approach verifies application functionality in real-world scenarios and is helpful in migrating legacy systems to modern architectures.

7.3 Consistency Validation Testing

Conducting consistency validation tests is crucial to ensuring a smooth transition from a legacy to a new system. These tests compare the new application's outputs

against expected results from the legacy system for the same input data. A pivotal example of this process is encapsulated in a concise test case shown in **Figure 20**.

Consistency Test

```
@SpringBootTest
public class ApplicationResultComparisonTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testResultConsistencyWithLegacyData() throws IOException {
        String testData = "sample data to process";

        // Process data in the new application
        ResponseEntity<String> response = restTemplate.postForEntity("/processData", testData, String.class);
        String resultFromNewApp = response.getBody();

        // Load expected result from a file (legacy application result)
        Path filePath = Paths.get("src/test/resources/expectedResult.txt");
        String expectedLegacyResult = Files.readString(filePath);

        // Assert that the result from the new application matches the expected legacy result
        assertEquals(expectedLegacyResult, resultFromNewApp, "The processed result from the new application
should match the legacy application's result.");
    }
}
```

Figure 20. Consistency Test.

The testing process entails providing the same input data to both the legacy system and the new application. Once the output is retrieved from the new application, it is compared against expected results, which are derived from the legacy system processing and stored in a file.

This approach is designed to not only verify that the migrated application functions equivalently to the original application, but also to build confidence in the accuracy and reliability of the migration process. The thorough testing ensures that the new system processes data in a consistent manner with the established legacy application. This highlights the critical role of testing in the success of software migrations.

8 CONCLUSIONS

Throughout this project, each step was strategically planned and executed to ensure a smooth transition from the legacy desktop application to a web-based platform. This migration journey was guided by several critical research questions, which helped shape the methodology and ensure that each decision was backed by thorough analysis and strategic foresight.

The initial research question focused on identifying effective methodologies to analyse the legacy application's codebase, integrations, and user interface. This was crucial for setting a solid foundation for the migration process. By employing high-level analysis and detailed examination of the codebase, invaluable insights were gained into the operational intricacies of the legacy system. These analyses provided a clear vision for the development process and ensured that the transition strategy aligned with the system's architectural and functional realities. Chapter **6** detailed the findings from this phase, which laid out the blueprint for the subsequent stages of development.

Another pivotal aspect of the project was addressed in the chapter **6.6**. Here, the best practices for migrating data from desktop-based MS Access to a centralized Oracle Database were explored. This chapter answered the question regarding best data migration practices and demonstrated the practical application of these methodologies, contributing significantly to the project's success.

In the chapter 5, the discussion centred around choosing the most suitable technology stack for the new web-based application. This selection was meticulously aligned with the capabilities of the legacy system and the anticipated future scalability needs. The choice of Java, Spring Boot, Angular, and Oracle Database, supported by Docker and Rancher for deployment and orchestration, proved to be not just practical but also forward-thinking, ensuring the new system's long-term viability.

Finally, the chapter 7 highlighted the strategies to validate the migrated system. Utilizing Spring Boot with JUnit 5 enabled comprehensive testing, from unit and integration tests to performance and functionality validations. This rigorous testing ensured that the new web application not only met but, in many aspects, exceeded the performance and functionality standards of the legacy system.

In conclusion, the success of the project was greatly aided by the methodologies chosen, technologies implemented, and testing strategies used. This migration has improved the capabilities of the system and positioned it for success in an increasingly digital future. The project has demonstrated significant achievement in both academic and practical realms.

As this thesis is being written, the first version of a new web-based application had successfully transitioned from the legacy system, embodying the functionalities of the original application within a modern framework. This milestone marks only the beginning of an ongoing journey, as plans are already underway to enrich the application with additional integrations and functionalities at the behest of the client. The primary objectives have been met: customer has a contemporary, web-based application designed for ease of future updates and enhancements.

In the future, Wapice Ltd will continue to maintain and develop the new application. This ensures that the service will remain uninterrupted, and the application will evolve to meet the changing needs and technologies. Migration can be considered a successful project that has laid a strong foundation for future innovations and enhancements.

REFERENCES

/1/ Gartner. (n.d.). Legacy Application or System. Accessed 16.2.2024. https://www.gartner.com/en/information-technology/glossary/legacy-application-or-system

/2/ Talend. (n.d.) What is a Legacy System. Accessed 16.03.2024. https://www.talend.com/resources/what-is-legacy-system/#:~:text=A%20legacy%20system%20is%20outdated,all%20it%20will%20ever%20do.

/3/ United States GAO. 17 January 2019. Accessed 16.2.2024. https://www.govinfo.gov/content/pkg/BUDGET-2019-PER/pdf/BUDGET-2019-PER-7-3.pdf

/4/ Gillin, Paul. 21 June 2021. Why the Cost of Maintaining Legacy Systems Only Grows Over Time. Accessed 20.02.2024. https://www.mendix.com/blog/why-the-cost-of-maintaining-legacy-systems-only-grows-over-time/

/5/ Pavel Kovalenko. 9 April 2021. The Cost of Maintaining Legacy System: How Much You Overpay. Accessed 20.02.2024. https://modlogix.com/blog/the-cost-of-maintaining-legacy-systems-how-much-you-overpay/

/6/ Warwick Ashford. Legacy IT systems a significant security challenge. 6 June 2019. Accessed 20.02.2024. https://www.computerweekly.com/news/252464666/Legacy-IT-systems-a-significant-security-challenge

/7/ Sieuwert van Otterloo. Technology risk assessment 2023: what programming language (versions) to use. 13 January 2023. Accessed 20.02.2024. https://ictinstitute.nl/technology-risk-assessment-2023-what-programming-language-versions-to-use/

/8/ Altexsoft. Legacy System Modernization: How to Transform the Enterprise for Digital Future. 30 December 2019. Accessed 20.02.2024. https://www.altexsoft.com/whitepapers/legacy-system-modernization-how-to-transform-the-enterprise-for-digital-future/

/9/ Andrew Nechiporuk. 10 Reasons to Upgrade Your Legacy CRM and Move to he Cloud. 6 March 2020. Accessed 20.02.2024. https://www.nextiva.com/blog/legacy-crm.html

/10/ David McCarty. What Are the Biggest Problems with Legacy Software? (n.d) Accessed 20.02.2024. https://www.gavant.com/library/what-are-the-biggest-problems-with-legacy-software

/11/ Sharat Chandler. The Arrival of Java 21. 19 September 2023. Accessed 07.03.2024. https://blogs.oracle.com/java/post/the-arrival-of-java-21

/12/ Pooja Sharma. Essential Spring Boot Concepts. 18 December 2023. Accessed 07.03.2024. https://blogs.perficient.com/2023/12/18/part-2-essential-spring-boot-concepts/

/13/ Spring Data. Accessed 02.03.2024. https://spring.io/projects/spring-data

/14/ Hibernate. Accessed 02.03.2024. https://hibernate.org/

/15/ Minko Gechev. 15 February 2024. Angular v17.2 is now available. Accessed 07.03.2024. https://blog.angular.io/angular-v17-2-is-now-available-596cbe96242d

/16/ Solarwinds. What Is an Oracle Database. (n.d.). Accessed 07.03.2024. https://www.solarwinds.com/resources/it-glossary/oracle-database

/17/ Praven Dandu. Understanding Docker Architecture. 8 August 2023. Accessed 05.03.2024. https://praveendandu24.medium.com/understanding-docker-architecture-an-in-depth-overview-of-docker-components-and-usage-f1a26bd217f9

/18/ Rajesh Kumar. What is Rancher and How it works? An Overview and Its Use Cases. 21 March 2022. Accessed 07.03.2024. https://www.devopsschool.com/blog/what-is-rancher-and-how-it-works-an-overview-and-its-use-cases/

/19/ DBeaver community. Accessed 02.03.2024. https://dbeaver.io/

/20/ MEAN vs Spring-Boot. Accessed 30.04.2024. https://stackshare.io/stackups/mean-vs-spring-boot

/21/ .NET vs Spring Boot. Accessed 30.04.2024. https://stackshare.io/stackups/dot-net-vs-spring-boot