

A modular framework for the effective development of web-based applications

M.C. van der Bank

 **orcid.org 0000-0002-4431-8577**

Dissertation submitted in fulfilment of the requirements for the
degree *Master of Engineering in Computer and Electronic
Engineering* at the North-West University

Supervisor:

Dr J.C. Vosloo

Graduation May 2018

Student number: 28344707

Abstract

Title: A modular framework for the effective development of web-based applications
Author: MC van der Bank
Supervisor: Dr JC Vosloo
Keywords: Web framework, technical debt, modular software, rapid application development, MVC architecture, efficient maintenance

A significant part of the software development life cycle consists of maintenance. Software maintenance is crucial to ensure that bugs are fixed, and continuous updates are applied to increase performance. Regular maintenance also ensures that technical debt is repaid and kept to a minimum. Technical debt refers to the concept where compromises are made to code quality and performance in order to achieve a shorter software release cycle. With only a finite amount of development resources available, large amounts of technical debt may lead to various types of problems at a later stage.

This study introduces a solution in the form of a web framework. The framework is based on the Model-View-Controller (MVC) architecture. The modular nature of the framework allows for Rapid Application Development (RAD) and ensures that maintenance can easily be performed. A set of rules for best-practices with regards to implementation is provided in order to reduce technical debt.

A number of case studies were implemented using this framework to evaluate its effectiveness in terms of modularity, scalability, ease of maintenance and rapid development. The specifications of each case study were defined by the specific need for an application that was required by an Energy Services Company (ESCO). The applications implemented in the case studies either aided the ESCo personnel in day-to-day activities, or provided tools with which to manage other existing systems.

Software engineers employed by the ESCo were asked to complete a survey with regards to experiences gained by developing for case studies on the framework. These engineers all worked on an existing web framework owned by the ESCo, and were able to compare the two frameworks in terms of learning curve, available resource material, ease of development, code quality, maintainability, technical debt accrued and best practices followed by each framework.

From the results yielded by the survey, it was observed that the engineers rated the framework developed in this study higher in all questions asked when compared to the existing framework. Out of

the maximum score of 5 points (where higher is better), this framework scored an average of 4.42, as opposed to a score of 2.73 for the existing framework.

Technical debt was analysed in more detail by using a software tool called SonarQube, which uses several metrics to quantify the amount of technical debt. These include bugs, vulnerabilities, debt, code smells and duplications. From the analysis it was observed that this framework only consisted of 9.2% duplicated code and 15 days of technical debt. In contrast, a system of similar size used by the ESCo had 23.6% duplications and 206 days of technical debt.

It was concluded that the framework developed in this study achieved all of its objectives. Modular components were developed, which provides reusable functionality to all applications developed on the framework. The survey proved that efficient development was obtained through the use of modular components, rigid coding standard and best-practice procedures and guidelines. Finally, technical debt present in the framework components was measured to effectively be less than the debt in the overall web platform, which include applications developed on the framework.

Acknowledgements

First and foremost, thank you Lord Jesus Christ for your love, patience and blessings. Without you none of this would have been possible.

Furthermore, I would like to thank the following people:

- My parents, for their love and support during my studies
- Dr. J.C. Vosloo, Dr. J.N. du Plessis and Dr. S.W. van Heerden, for their guidance, insights and support
- Prof. E.H. Mathews and Prof. M. Kleingeld, for providing the opportunity to further my studies at CRCED Pretoria

Lastly, I would like to thank TEMM International (Pty) Ltd for providing funding for the research and development of this dissertation.

Table of Contents

Abstract	i
Acknowledgements	iii
Table of Contents	iv
List of Figures.....	vi
List of Tables.....	ix
List of Abbreviations.....	x
1. Introduction.....	1
1.1. Preamble	2
1.2. Background: Challenges in Software Development	2
1.3. Need for the Study	7
1.4. Problem Statement	8
1.5. Objectives	8
1.6. Document Overview.....	8
2. Literature Review of Existing Solutions	10
2.1. Preamble	11
2.2. Software Development Methodologies	11
2.3. Challenges of Software Development.....	20
2.4. Software Distribution	28
2.5. Web Development	33
2.6. Summary of the Literature	36
3. Design of a Modular Framework.....	37
3.1. Preamble	38
3.2. Framework Requirements.....	38
3.3. Development Software and Systems	40
3.4. Technical Design	42
3.5. Additional Features	53
3.6. Development Practices.....	58
3.7. Framework Verification.....	61
3.8. Design Summary.....	67
4. Results and Case Studies	68
4.1. Preamble	69

4.2.	Modularity	69
4.3.	Case Study	76
4.4.	Developer Survey	86
4.5.	Survey Results	90
4.6.	Technical Debt.....	94
4.7.	Summary of the Results	97
5.	Conclusion and Remarks	98
5.1.	Preamble	99
5.2.	Summary of Work Completed.....	99
5.3.	Study Conclusion	100
5.4.	Limitations and Future Scope.....	101
6.	References.....	103
7.	Appendices	108

List of Figures

Figure 1. Dependency hierarchy	3
Figure 2. Software client types.....	4
Figure 3. ESCo software structure	6
Figure 4. Waterfall methodology process	12
Figure 5. SCRUM methodology process	14
Figure 6. RAD methodology process	14
Figure 7. PEP methodology process	16
Figure 8. Combining waterfall and SCRUM processes	17
Figure 9. Improved requirement engineering process.....	18
Figure 10. Adapted RAD process	19
Figure 11. Technical debt visualisation	22
Figure 12. Technical debt dependency hierarchy	23
Figure 13. Adapted waterfall SDM	24
Figure 14. Iterative Adaptive Life-cycle process.....	25
Figure 15. Technical debt metrics	25
Figure 16. Client-server model	29
Figure 17. Client-server load balancing.....	29
Figure 18. Decentralised distribution model.....	29
Figure 19. MVC architecture	35
Figure 20. Multi-platform web-based user access.....	39
Figure 21. Model binding	43
Figure 22. MVC process overview	43
Figure 23. Extended MVC process overview	44
Figure 24. Data Access Layer	45
Figure 25. VM example.....	46
Figure 26. Platform structure overview	47
Figure 27. Area and namespace hierarchy	47
Figure 28. Framework structure overview	48
Figure 29. Application structure overview	49
Figure 30. DAL Inheritance	50
Figure 31. Layout inheritance.....	52
Figure 32. General page layout	52
Figure 33. Authentication process	54

Figure 34. User right database schema	55
Figure 35. Custom error handling	57
Figure 36. Guidelines for adding a web application.....	59
Figure 37. User login page.....	62
Figure 38. Successful user login attempt	62
Figure 39. Failed user login attempt	63
Figure 40. Successfully updated last login date	63
Figure 41. Manually rewind last successful login date.....	63
Figure 42. Mobile browser layout	64
Figure 43. Desktop browser layout	65
Figure 44. User right that grants access to the dashboard	65
Figure 45. User right that grants access to application 1.....	66
Figure 46. Access URL without logging in.....	66
Figure 47. Access URL with insufficient rights.....	66
Figure 48. Unauthorised access error	67
Figure 49. Existing user interface example	70
Figure 50. Example data	70
Figure 51. Add the new column in MySQL	71
Figure 52. Add a new property to the Model.....	71
Figure 53. Add SQL mapping to DAL.....	71
Figure 54. Add SQL statement to DAL	72
Figure 55. Add column on View	73
Figure 56. Updated user interface with new column.....	73
Figure 57. Standard architecture.....	74
Figure 58. Modified architecture for Microsoft SQL Server	75
Figure 59. Modified architecture for MongoDB.....	75
Figure 60. Application A framework component usages	78
Figure 61. Lines of code distribution in Application A.....	79
Figure 62. Application B framework component usages	81
Figure 63. Lines of code distribution in Application B.....	82
Figure 64. Application C framework component usages	84
Figure 65. Lines of code distribution in Application C.....	85
Figure 66. Developer survey results	90
Figure 67. Developer survey section averages.....	91
Figure 68. Normal distribution of scores per platform	91

Figure 69. Detailed technical debt analysis to compare platforms.....	96
Figure 70. Technical debt analysis of core components vs. entire platform	96

List of Tables

Table 1. Selection of Google products	32
Table 2. ASP.NET vs PHP performance evaluation	35
Table 3. Example database table.....	42
Table 4. Guidelines followed for Work Logger	77
Table 5. Application A code usage.....	79
Table 6. Guidelines followed for Database Manager	80
Table 7. Application B code usage.....	82
Table 8. Guidelines followed for Services	83
Table 9. Application C code usage.....	85
Table 10. Survey categories.....	87
Table 11. Survey questions standard deviation	92
Table 12. Analysis results overview.....	95
Table 13. Rating descriptions	95
Table 14. Validation methods	97
Table 15. Developer survey results	112
Table 16. Detailed analysis results	112
Table 17. Framework vs. platform technical debt analysis.....	113

List of Abbreviations

AHP	Analytical Hierarchy Process
API	Application Programming Interface
CD	Compact Disk
CMS	Condition Monitoring System
CRUD	Create Read Update and Delete
CSS	Cascading Style Sheet
DAL	Data Access Layer
DSDM	Dynamic Systems Development Model
EDS	Electronic Delivery of Software
ESCo	Energy Services Company
FDD	Feature-driven Development
GA	Genetic Algorithm
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IIS	Internet Information Server
IoT	Internet of Things and Services
IT	Information Technology
JS	JavaScript
LPLC	Linear Predictive Life Cycle
MSF	Microsoft Solutions Framework
MVC	Model-View-Controller
NPM	NuGet Package Manager
OOP	Object Orientated Programming
OPC UA	Open Platform Communications Unified Architecture
PEP	Product Evolution Process
RAD	Rapid Application Development
RDBMS	Relational Database Management System
RUP	Rational Unified Process
SaaS	Software as a Service
SDM	Software Development Methodology
SOAP	Simple Object Access Protocol

SQL	Structured Query Language
TDD	Test-driven Development
URL	Uniform Resource Locator
VM	View Model
VoIP	Voice over Internet Protocol
WaaS	Windows as a Service
WSDL	Web Service Definition Language
WSN	Wireless Sensor Network
WSRA	Web Service Reference Architecture
XML	Extensible Markup Language
XP	Extreme Programming

Chapter 1

Introduction

1.1. Preamble

This chapter will provide background information on the topic concerning this paper. This includes software maintenance, resulting technical debt from neglected maintenance, the problem concerning limited development resources and the distribution of software.

The need for the study is justified in terms of various problems relating to software development concerning both the development process and distribution of software. Lastly, a problem statement is defined, and goals are set by which this study aims to solve the problem at hand.

1.2. Background: Challenges in Software Development

Software Maintenance

Software development is an evolutionary process. Software is constantly changing and adapting to the current needs of the ecosystem in which it resides. A software ecosystem is defined by Lungu [1] as “*a collection of software projects that belong to an organisation and are developed in parallel by the organisation*”. Eick et al. [2] observed that software decays over time, and that maintenance becomes increasingly expensive. They define code decay as “*if it is more difficult to change than it should be*”, in terms of cost, interval and quality. It is therefore essential that a software ecosystem is maintained in order to improve performance and continuously add new functionality.

Maintenance on software can be a time-consuming and complex exercise, especially if multiple developers were involved on a project. One of the most complex areas to maintain in software ecosystems are dependencies [3]. If a dependency package is updated, all other packages that rely on it might be affected due to functionality that might have changed slightly. One such example occurred in 2016 when the *leftpad* package was removed from the popular online package repository, *npm*, causing many web development projects to fail¹. Even though this package only consisted of 11 lines of code, it was relied upon by large projects such as *React*², used by Facebook and *Node*³.

Figure 1 illustrates an example of package dependencies, where component A may be considered as the user interface of a web-site. If package H is removed, or fails to compile, all higher-level packages, including component A, will fail. This is due to the fact that an output, or computational result from package H may be passed up the hierarchy to the top most package or component.

¹ <https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/>

² <https://facebook.github.io/react/>

³ <https://nodejs.org/en/>

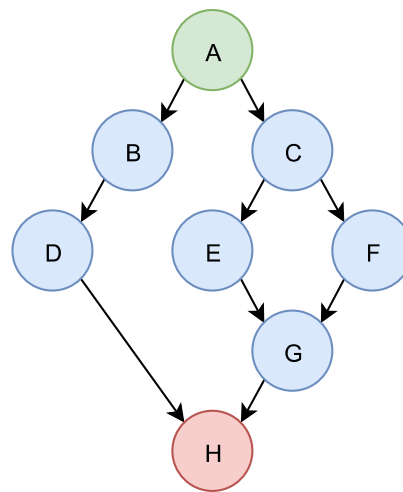


Figure 1. Dependency hierarchy

According to Lin et al. [4], approximately 80% of a software company's budget is spent on maintenance. It is therefore crucial that software is designed with maintainability in mind, as well as optimisation of cost and labour associated with maintenance.

Technical Debt

Technical debt is a term first used by Cunningham [5] in 1992, and explains how metaphorical debt is incurred when a short software release cycle is prioritised over software quality. For example, consider the scenario where new functionality should be added to an existing software system. If achieving the target release date is considered more important than thorough testing and quality assurance, the software system is considered to be in technical debt. It would require more development time and resources at a later stage to ensure that the initial release of the software is brought up to an acceptable standard.

Technical debt may similarly be incurred if a system is developed or expanded with temporary functionality and with the intent to implement a permanent solution at a later stage. As time progresses, and with development resources continually being dedicated to new functionality, maintenance on the temporary code may be neglected. In an even more concerning scenario, new functionality may come to depend on the temporary implementation. When the initial release is then finally updated to a permanent solution, all other dependents should be maintained as well.

It is therefore clear that an increase in technical debt will result in higher maintenance activities [6]. The compromise between design quality and time-to-market may lead to increased financial overhead, according to Ampatzoglou et. al. [6]. It was estimated that the global technical debt in 2010 was \$500 million, which could be doubled in five years' time.

Limited Resources

Irrespective of the size of an organisation, resources are always in limited supply. This holds particularly true for human resources allocated to software development. From a business perspective, it might make more sense to invest resources into new feature-driven developments and to meet short-term objectives. Unfortunately, stakeholders (without software engineering expertise) involved in these decisions are not always fully aware of the long-term implications of technical debt [7].

Software Distribution

Early software for microcomputers and video-game consoles used to be distributed by means of physical copies in a packaged form [8]. This included floppy disks, cartridges and later compact disks (CDs). However, during the 1970s and 1980s, electronic delivery of software (EDS) services began to commercialise. EDS was based on the concept of utilising a communications network to deliver software electronically and directly to the consumer.

Software distribution can mainly be achieved through *fat clients* or *web clients* [9][10]. A fat client is a desktop application that has to be downloaded and installed on an end device in order to run, whereas a web client application can be accessed through a web browser. The main difference between the two types of clients is that fat clients are able to run offline and in isolation, while web clients require an internet connection to access.

Figure 2 provides a visualisation of fat- and web clients. The majority (or all) of a fat client's code is localised to the user's device. The application may optionally connect to the internet and download updates. Since a web client is accessed via the internet, all of its application code is located on a remote server.

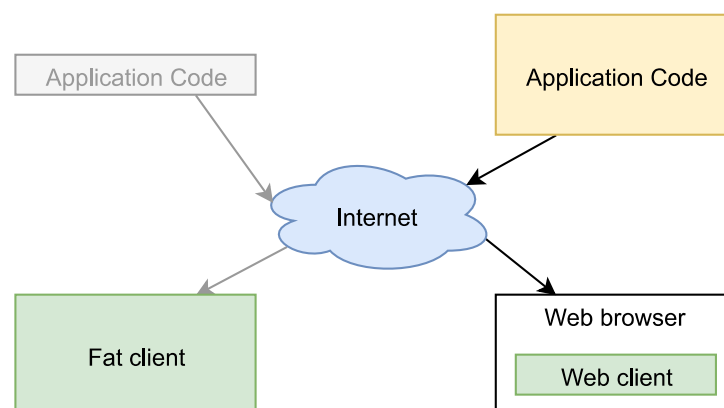


Figure 2. Software client types

In order to keep desktop software updated, the user (or desktop software itself) will have to periodically check if a newer version is available. Software updates are then downloaded from a central server in the

client-server architecture [11] prior to being installed. In some cases, if desktop software is out of date, there is a risk that the software may cease to function, especially if the software uses some form of network communication. This may be due to changes in communication protocols, which would require both client and server applications to be up to date.

The advantage of using web client applications over traditional desktop software is that the web-based application will always be up to date from the client's perspective, since the application's executable code is located on a remote server. However, a drawback is that web clients are reliant on an internet connection.

Another form of online software distribution is Software-as-a-Service (SaaS) [12]. SaaS provides consumers with access to online applications or resources, which is hosted and maintained by its provider. The provider may typically charge a subscription fee for access to its services.

Software in the Industry

There are many challenges involved in various areas of software development, including development, deployment and maintenance. In a case study conducted by Mantyla and Vanhanen [13], the software deployment activities and strategies for four companies were examined. Three characteristics were identified that increased the difficulty of product deployment:

- The level of integration with other software systems.
- The complexity of various configurations needed in order to use the software.
- The requirement of real-world data in order to use the software.

The above case study further found that the most important goals for software deployment were identified as the need to decrease the level of expertise required to deploy, and decrease the deployment effort. If a high level of expertise is required to deploy a product, a smaller number of personnel will be able to assist in product deployment. Companies therefore aim to increase knowledge transferred among their employees. Factors that contribute to deployment effort include the following:

- The amount of time needed to configure the product.
- Product configuration is a collaborative activity between the vendor and customer, therefore a formal process is needed.
- Building interfaces to integrate the product with existing systems is a laborious task.
- Obtaining proper data from the customer to create a model took considerable effort.

In a separate case study done for this paper, an investigation was done for an Energy Services Company (ESCo) on how they utilise software for internal management and activities. The company, hereafter referred to as the ESCo, mainly provides its clients with services to promote energy savings. As a value-added service, the ESCo provides its clients with an online energy dashboard. This dashboard gives overviews on current and historical energy usages, estimated cost savings and automated report generation.

However, for the dashboards to be able to show data, an immense amount of configuration and data processing is required. The ESCo uses a collection of internally developed desktop applications to be able to retrieve and process data, and set up the dashboards for the online front-end interface. Each desktop application is maintained as its own “project”, and often requires updates to fix bugs or add functionality. Figure 3 illustrates the structure of the ESCo’s software suite. The software suite, which is installed on each company workstation, consists of the various desktop applications. These applications connect to the online dashboards and web interfaces through the internet.

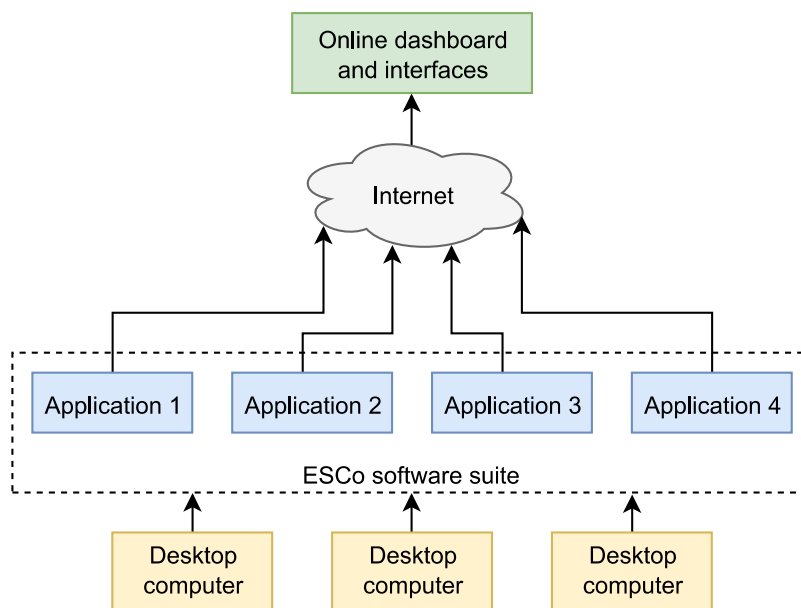


Figure 3. ESCo software structure

Whenever an update is released for an application, all the ESCo personnel should manually update that application on their workstations by downloading the newest release from the company’s local network. This fragmentation of software projects requires a significant amount of the ESCo’s development resources, since each application should be maintained individually. Since each application works differently from the rest, one or two developers should specialise in that application. This means that it is harder for the developers to consult on other applications outside of their expertise, and creates a problem if a specific developer leaves the ESCo, or is assigned to another project.

1.3. Need for the Study

A few problems with regards to software development have been identified in the previous section. Some of these issues are related to the ineffective management of software engineering, whilst others are more technical in nature. These problems are discussed below in order to justify the need for the study.

Inefficiencies and problems related to various aspects of software development have the potential to cause numerous issues or delays. Due to pressure to rapidly release software, some compromises might be made which affect the overall quality of the software. This then leads to an increased need for maintenance, and may even cause maintenance to be more complex. This increased need for maintenance, or temporary software implementations for the sake of quickly releasing software, incurs technical debt. Only a limited number of development resources is available to address the need for maintenance.

It may be argued that incurred technical debt also collects interest. If a temporary feature that was implemented is a dependency for other functional parts of the software product, those parts are also considered to be in debt. This is due to the fact that, when the original temporary feature is correctly implemented, the features depending on it may also need to change in order to work correctly.

In order to repay the technical debt, it will be necessary to allocate development resources. This costs additional time, money and effort, which could have been spent on new development. In severe cases, projects with an overwhelming amount of technical debt may fail, which may result in financial loss. It is therefore imperative that the incursion of technical debt be managed according to release schedules and available development resources.

In addition to the technical issues of developing a software project, the practical challenges, such as deploying and distributing a software product, may also cause difficulties. The current software distribution trend is to distribute it via electronic means over the internet. However, if software is physically installed on a user's computer, regardless of the method of acquisition, the problem of keeping that instance of the software up to date arises and applies to the entire user base.

By distributing software online or as a service, only a single deployment of the application on the web is required. The issue of keeping installations up to date on users' devices is then eliminated. When a user accesses the web application via any web browser (be it mobile or desktop), the most up to date instance of the application is presented to the user.

From these issues, the need exists for a centralised web framework. This framework should provide employees, such as the ESCo's personnel discussed in section 1.2, with an online set of management tools and utilities, while at the same time simplifying development. Maintenance should be simplified and less time consuming if the framework minimises technical debt, thereby lessening the strain on development resources.

1.4. Problem Statement

Software development projects may become inefficient due to various problems encountered throughout the development life cycle. These may include the accumulation of technical debt and the issue around maintenance. If an insufficient number of development resources is available, inadequate maintenance may be performed, which may lead to technical debt not being paid off.

1.5. Objectives

The aim of this study is to improve software development efficiency by developing a framework with the following characteristics:

- Must be modular by providing simple and reusable components, which can be used to rapidly develop new functionality.
- Ensure that maintenance may be performed in an efficient manner.
- Minimise the accumulation of technical debt to lessen the strain on development resources.
- Be considered an acceptable development framework by software developers.

1.6. Document Overview

The rest of this document is divided into the following chapters:

Chapter 2 – Literature review

A comprehensive literature review is conducted on existing systems and solutions addressing the problems discussed in this chapter. The literature review investigates the importance of software development methodologies, and the role these play in the effective development of software projects. The problem regarding technical debt is further discussed, and case studies for technical debt mitigation

are presented. The literature also reviews historic and current methods for software distribution. Finally, an investigation was done on various web-based technologies and architectures, and systems that implement these.

Chapter 3 – Design

By using concepts and methods discussed in chapter 2, a complete and detailed solution is presented to solve the problems identified in chapter 1. A list of framework requirements is defined, followed by a brief overview of the development environment that was used. The technical design is divided into two subsections: a section that defines the core of the framework, and another section that discusses additional framework features such as security, routing and error handling. It was lastly verified whether or not the framework met the specifications that were set.

Chapter 4 – Results

Validation is performed to ensure that the developed framework solved the problems that were identified in chapter 1. The framework is validated by means of a case study, developer survey and technical debt analysis. The case study implements a selection of web applications, and evaluates the framework's effectiveness in terms of modularity and reusable components. The survey attempts to quantify the effectiveness of the framework from the perspective of software developers. Technical debt in the framework is also analysed and compared to that of an existing web system.

Chapter 5 – Conclusion

The study is concluded by summarising the work that has been done in this paper, and draws overall conclusions from the results that were obtained. Comments are lastly made on the limitations of the framework developed in this paper, and how it may be improved in the future.

Chapter 2

Literature Review of Existing Solutions

2.1. Preamble

This chapter presents a literature review on solutions for problems which were identified in chapter 1. An investigation into software development methodologies is made, which formulates a process by which software projects are conducted. Various traditional and “textbook” methodologies are highlighted, as well as case studies from the industry where these methodologies were modified or combined in specific environments.

After the importance of development methodologies are shown, the management and challenges of software development projects are discussed. Problems related to software development, such as maintenance and technical debt, is discussed, as well as models and solutions proposed to solve these problems.

Another aspect related to software development is the deployment and distribution thereof. A brief history of distribution methods is presented, and various modern-day electronic distribution models are described. The advantages and disadvantages of software distribution via the internet is also discussed.

The literature study concludes with a summary of various web-based technologies and architectures, and presents case studies of systems that implement these technologies. Critical elements and concepts that are discussed in this chapter, such as the Model-View-Controller architecture, are used to design a solution in chapter 3.

2.2. Software Development Methodologies

A Software Development Methodology (SDM) is a framework or process according to which a software project is developed. It describes a detailed structure of dividing the development process into smaller phases. Each phase is designed to optimise a certain aspect of the development process, such as planning, design, development, or maintenance. An SDM is also known as the software development life cycle. There are many well-known and less common SDMs in practice [14][15], including, but not limited to:

- Crystal
- Dynamic Systems Development Method (DSDM)
- Extreme Programming (XP)
- Feature-driven Development (FDD)
- Microsoft Solutions Framework (MSF)
- Rapid Application Development (RAD)
- Rational Unified Process (RUP)
- SCRUM
- Spiral
- Test-driven Development (TDD)
- Waterfall

The waterfall methodology is the traditional approach to software development, which involves linear and sequential processes [16]. This methodology focuses on early-stage planning and aims to eliminate problems before they occur. As shown in Figure 4, a new process is initiated upon the completion of the previous, without any iteration.

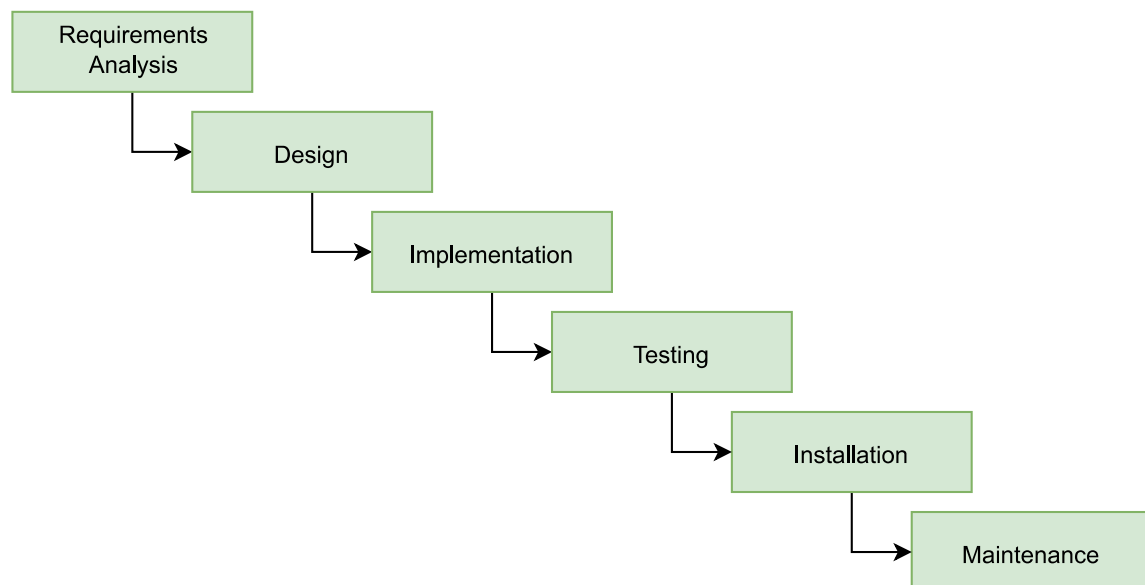


Figure 4. Waterfall methodology process

Agile software development describes a range of iterative, interactive and incremental development techniques and principles [17]. While Agile is not a methodology, its processes, as described in the Agile Manifesto⁴, is used to define Agile methodologies such as Crystal, DSDM, XP and SCRUM. The Agile Manifesto describes four aspects of software development that forms the moral basis of its methodologies. The manifesto reads as follows:

⁴ <http://agilemanifesto.org/>

*“We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:*

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

*That is, while there is value in the items on
the right, we value the items on the left more.”*

The manifesto further describes twelve principles of Agile software development that describes the characteristics for a methodology. These principles focus on the following areas:

- Continuous delivery – Software should be delivered early and frequently.
- Adaptability – Development should be able to quickly cope and adapt according to changing specifications and requirements.
- Collaboration – Developers should stay motivated and continuously collaborate with stakeholders.
- Self-organising teams – Development teams should be able to plan and reflect at frequent intervals to optimise development.

By following these principles, Agile methodologies aim to accelerate and optimise the development process.

SCRUM is a very popular and widely used Agile SDM [17][18], which typically focuses on shorter cycles for planning, development and feedback for smaller teams [17]. Developers divide their work into smaller and more manageable work cycles, called “sprints”, which typically last between two and four weeks. Each sprint aims to deliver a functional piece of the solution that can be used or evaluated by the client. Sprints are completed until the solution is fully functional. The SCRUM methodology also includes daily stand-up meetings, during which project feedback and progress are discussed. Figure 5 illustrates the SCRUM process.

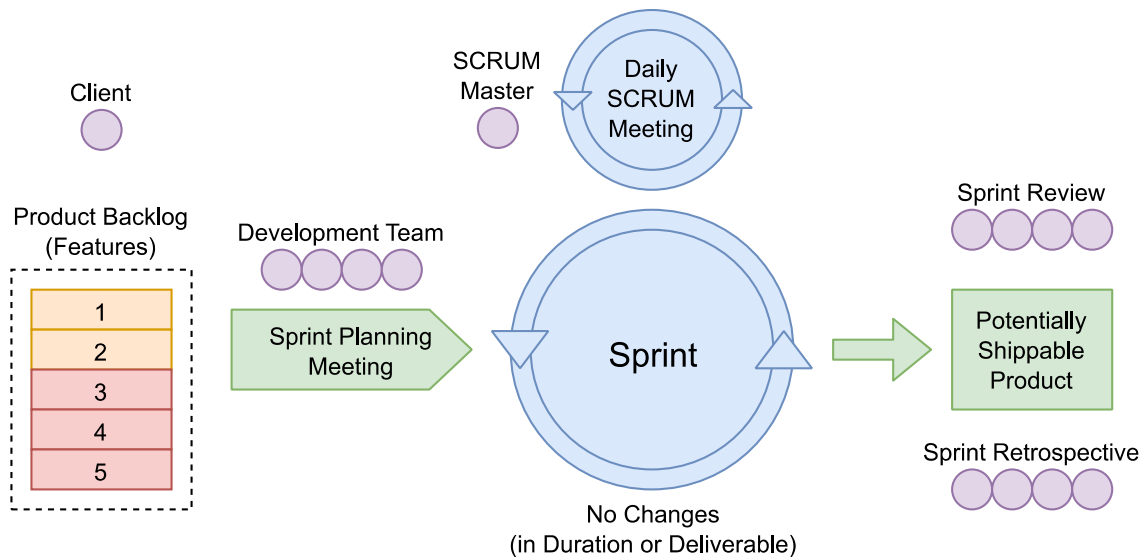


Figure 5. SCRUM methodology process

The product backlog is a list of requirements set by the product owner. During the initial product planning phase, the development team divides these requirements into various sprints. After each sprint has been completed, the deliverable, or functional feature is reviewed and its specification is adjusted, if necessary. After all sprints have been completed, the project is signed off and handed over to the client.

RAD is an iterative methodology that is designed to accelerate product delivery [19]. It places more emphasis on the development process, rather than on planning. The term RAD was first coined by James Martin in 1991 [20]. Martin wrote that "Rapid Application Development (RAD) is a development lifecycle designed to give much faster development and higher-quality results than those achieved with the traditional lifecycle. It is designed to take the maximum advantage of powerful development software that has evolved recently". Figure 6 illustrates the process of RAD.

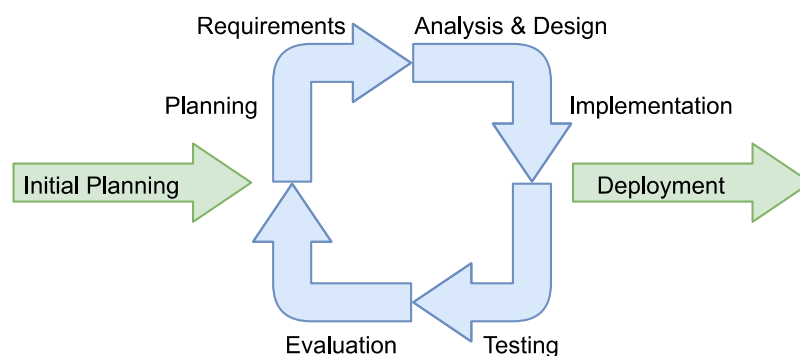


Figure 6. RAD methodology process

After initial planning has been performed, development commences by defining specifications and requirements for the product. Once the first implementation has been completed, the product is tested

and evaluated. Product specifications are adjusted based on new requirements, or feedback from the client. Product prototypes are often used instead of design specifications.

Khmelevshy et. al. [21] conducted a case study in which they presented findings from distributed Agile and SCRUM projects, and discussed the shortcomings of Agile. Some of the issues and recommendations discussed in the literature were:

- Many software development teams don't work from a single location, but in a distributed environment where there may be many teams, clients and locations involved.
- It is common practice for the Agile process to become "water-scrumfall" (waterfall methodology combined with SCRUM).
- SCRUM team size should ideally be between 10 and 15 people.
- Distributed teams should have equal workloads.
- To ensure that a high standard of quality is maintained, pair programming and regular code reviews should be conducted.
- Cultural and time differences should be taken into account.

The case study involved a software project where team members were spread across the US, Canada and Europe. During development, several issues were encountered, including inconsistent specifications and a lack of proper design. As a result, the project schedule had to be extended. The following conclusions and recommendations were made as a result of the study:

- Distributed projects lack the face-to-face communication element due to geographic differences. Alternative arrangements had to be made, such as Voice-over-IP (VoIP) teleconferencing.
- Avoid the "coding factory" approach where a specification is presented and a product expected. Agile processes should be implemented where all project stakeholders should collaborate.
- Plan for deliveries, demos and reflective meetings.
- Never start a project with incomplete or inaccurate specifications. The addition of high-level requirements at a late stage during development introduces numerous issues and wastes time and resources.
- Quality control during development is of the utmost importance.
- It is important to use the right development tools. A poor development environment may lead to low-quality software.

In another study conducted by Jha et. al. [18], the SCRUM methodology was scaled to an international level. Their methodology, Product Evolution Process (PEP), engaged 16 SCRUM teams with more than a thousand members across three continents. PEP combines the traditional waterfall methodology with SCRUM, as shown in Figure 7. This combination may also be referred to as “water-scrumfall”.

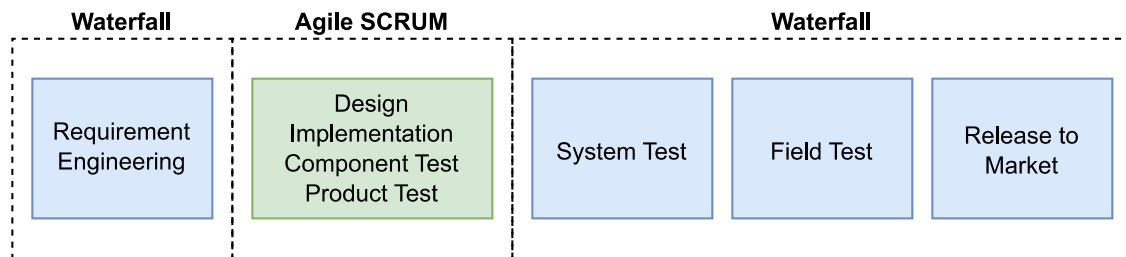


Figure 7. PEP methodology process

Since SCRUM is originally intended for smaller teams, several challenges were encountered where standard SCRUM practices were impacted. Face-to-face collaboration improves team communication and encourages the feeling of “one team”. However, since the developers, SCRUM master, client and other project stakeholders are not necessarily in the same time zone, communication becomes a challenge. Another issue encountered with scaling is that it becomes more challenging for the SCRUM master to coordinate a larger project and all involving factors, such as design, inter-team communication and dependency management.

By combining requirement engineering, field testing and release strategies from the waterfall methodology, with design, development and product testing from SCRUM, an optimal process was developed. Among the lessons learned from the study is that sprint coordination is crucial. Dependencies should be identified and taken into consideration when planning sprints for all teams. Furthermore, it was deemed important to find a balance between quality and feature implementation. Late changes to specifications or feature requests should also be avoided.

Traditional waterfall and SCRUM were also combined in another case study performed by Singhto and Phakdee [22]. The aim of the project was to develop specialised Software as a Service (SaaS) for small and medium enterprises in Thailand. The study argues that by combining the waterfall and SCRUM methodologies, a number of benefits were observed. These benefits include financial savings in terms of better time management, and increased customer satisfaction in IT services. Figure 8 illustrates the stages of the process.

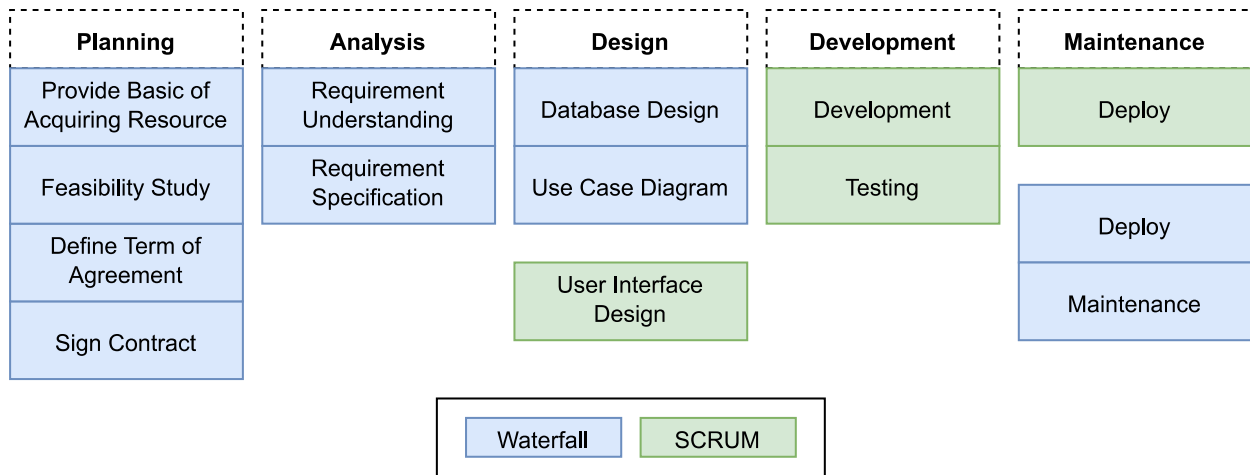


Figure 8. Combining waterfall and SCRUM processes

Both the planning and analysis phases utilise processes from the waterfall methodology in order to provide a specification and product backlog. During the design phase, use case diagrams are generated in order to understand the interactions between roles and systems. A database design is done to reduce the risk of project failures during later stages. Standard SCRUM processes are followed for further user interface design, development, testing and deployment.

It was concluded that the waterfall processes are easier to understand, especially for new or non-developers. Its linear approach aids with the identification of potential issues during the early stages of development. Since SCRUM is flexible in terms of specification changes, customers' needs will more likely be satisfied.

Hassan et. al. [23] noted that a large number of projects developed with the RAD methodology fails. The reasoning is that this is due to poor requirement engineering. Requirement engineering involves the process of defining problems and requirements, and is the first step in the waterfall methodology. They developed a seven-step model to improve requirement engineering for RAD, given in Figure 9.

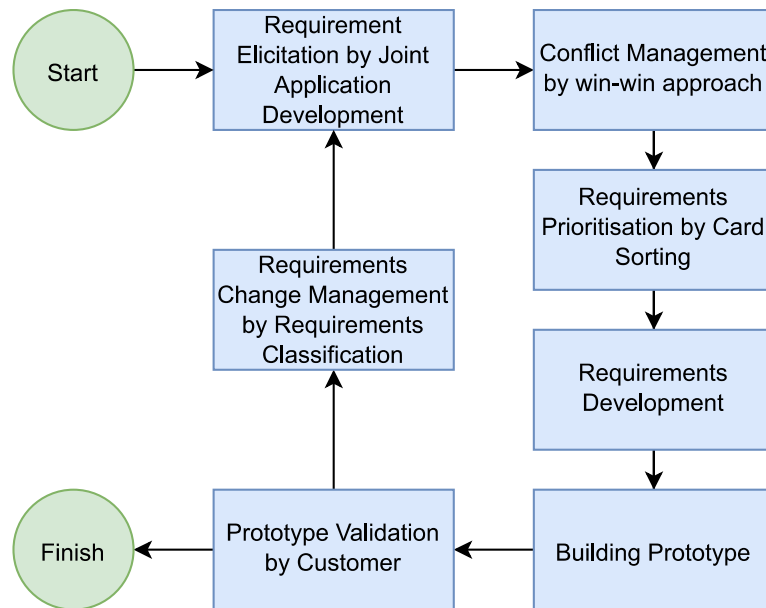


Figure 9. Improved requirement engineering process

The improved model was validated by means of survey forms, which were completed by 50 software engineers. The results indicated that 71.5% of the respondents were in favour of the newly proposed requirement engineering model. It was therefore concluded that their model was more time- and cost-effective than traditional methods, and achieved a high level of stakeholder satisfaction.

The RAD process may also be optimised by adapting waterfall processes. Prashanth [24] noted that a few problems exist when applying RAD on large-scale projects. In order to reduce overall development time, automatic code generation tools and reusable components are employed. This will help to minimise the effort required during the testing phase. However, it was concluded that developing reusable components for large-scale projects is difficult. If system components are not designed to be generic and reusable in multiple scenarios, new components will have to be developed for each specific requirement.

To address the issue of component reusability, the RAD process is adapted to better define specifications for components during the design phase. It requires the client to provide a complete set of requirements, which allows components to be designed with the full scope in mind. Figure 10 shows the adapted RAD process.

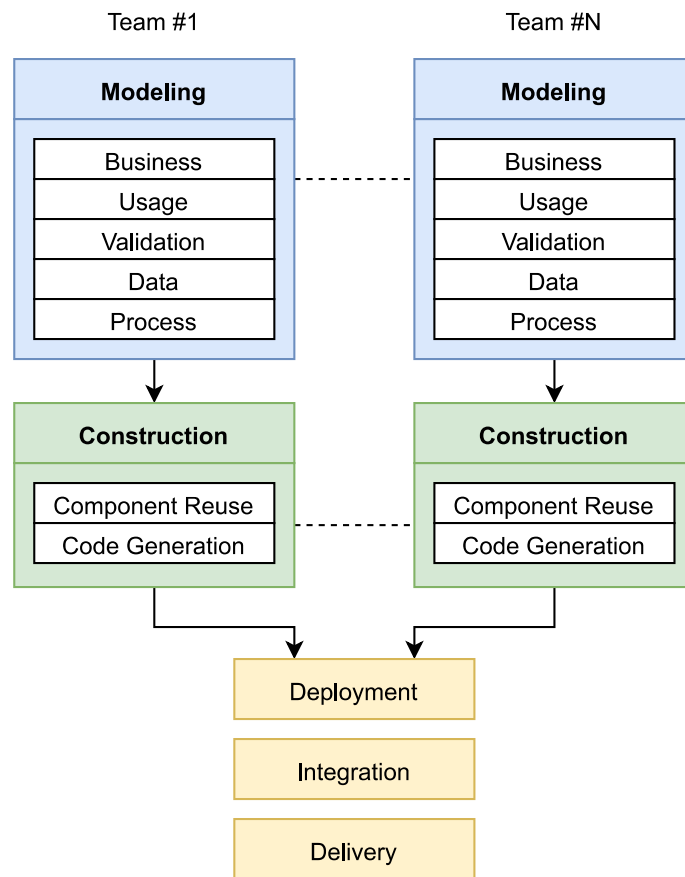


Figure 10. Adapted RAD process

The modelling phase defines when specific design stages take place. When all planning has been concluded, components are constructed and reused. Different teams are also tasked to work concurrently and independently. After all teams have finished their development, the solution is tested and delivered.

Summary

From all the studies analysed in this subsection, a few common shortcomings are apparent. Firstly, most iterative methodologies lack a clear stage where product specifications are defined. This may be due to the fact that these methodologies allow for specifications to change during the development life cycle. It may be argued that effort is not put into defining a complete specification during the early stages of development, since the client is allowed to change it, or request additional features during development. However, this may lead to delays, since completed functionality may need to change in order to work with the specification changes.

Another issue that was identified is the challenge of applying some methodologies to distributed development teams. Methodologies such as SCRUM or RAD were originally intended for small teams that can communicate face-to-face during the development life cycle. With teams that need to collaborate internationally, the cultural- and time zone difference becomes an issue.

From the various case studies where standard methodologies were adapted or modified, the following observations have been made:

- A single standard SDM is usually not sufficient for larger development teams. Every use case will have its own unique challenges that conflict with the standard SDM process.
- In order to suit an organisation's needs, a methodology may be modified, or even combined with another methodology.
- A clear product specification is required in order to effectively plan and develop the product. Although specification changes are allowed during the development life cycle, it is crucial to the success of the product to have a clear direction from the start.
- By using reusable components, products may be developed faster and more efficiently.

2.3. Challenges of Software Development

Since software development is an evolutionary process, it is logical to assume that some aspects of a developed product or service will change over time. This may be due to changing requirements or continued maintenance that corrects existing issues.

Eick et. al. [2] defined decayed code as code that is *“more difficult to change than it should be”*, and conducted a study on an existing software system consisting of more than 100,000,000 lines of code. According to them, there are multiple factors involved that may cause code to decay:

- An inappropriate architecture that does not support the required system changes.
- Inaccurate specifications that prevent developers from implementing the “right” system from the start.
- Time constraints that cause developers to make compromises in quality and stability in order to meet deadlines.
- A sub-optimal organisational environment, where issues such as low morale or ineffective communication may lead to low-quality work.
- Development experience, where, for example, a junior developer is unable to understand code written by a more senior and skilled developer.

There are furthermore three key factors described that play a role in the maintenance of decayed code. The first is the *cost* to the organisation in the remuneration of the developers, then the *interval*, which is

the period of time required to perform maintenance. Finally, the *quality* of the code after maintenance, which has bearing on the likelihood of the need to maintain the code again in the future. While the study did not specifically highlight the real-world implications of decayed code, it can be concluded that an increased development cost may be observed due to ongoing maintenance.

An extensive study conducted by Lenarduzzi et. at [25] investigated how software maintenance models evolved over the past 40 years. They analysed 78 peer-reviewed papers published between 1970 and 2015 that discussed models for assessing or predicting software maintenance. The following observations were made:

- Most models were developed from first principles without consideration towards existing ones.
- None of the models were validated from a third-party perspective.
- Most of the studies were based on private data inaccessible to other researchers, which makes it difficult to replicate results.
- An increase in maintenance research over time was observed. However, instead of consolidating models and maintenance processes, more models are being proposed.
- Despite the increase in research and proposed models, no new or novel metrics are being used.

This research indicates that, due to a lack of variety in maintenance models, yet more research is required in order to develop a reusable model for everyday use in the industry.

One method of approaching software maintenance is to keep record of a list of software defects. These defects are reported by users of the software system and are prioritised by the development team. Prioritising may be conducted by evaluating the severity of the issue and the impact it might have on the user base. If an issue is deemed to have an effect on a widespread number of systems, but not necessarily cause critical problems, it may still be classified as top priority.

A method was developed by Srewuttanapitikul and Muengchaisri [26] that uses natural language processing to analyse user feedback and prioritise identified issues. Certain keywords are extracted from user feedback forms and is then analysed against three impact factors: number of occurrences, severity and priority. An Analytical Hierarchy Process (AHP) algorithm is then used to rank issues, based on the three impact factors.

As termed by Cunningham [5], technical debt explains how metaphorical debt is incurred when a short software release cycle is prioritised over software quality. Technical debt may also be represented as

the difference in scope between the planned product and the delivered product (see Figure 11 for a visualisation of technical debt). This difference is work that will either delay the product delivery date (as in Figure 11), or have to be completed at a later stage (post-product delivery). If it is understood how a software project can incur technical debt, SDMs and maintenance models may be adjusted accordingly in order to minimise the debt incursion and effectively manage it.

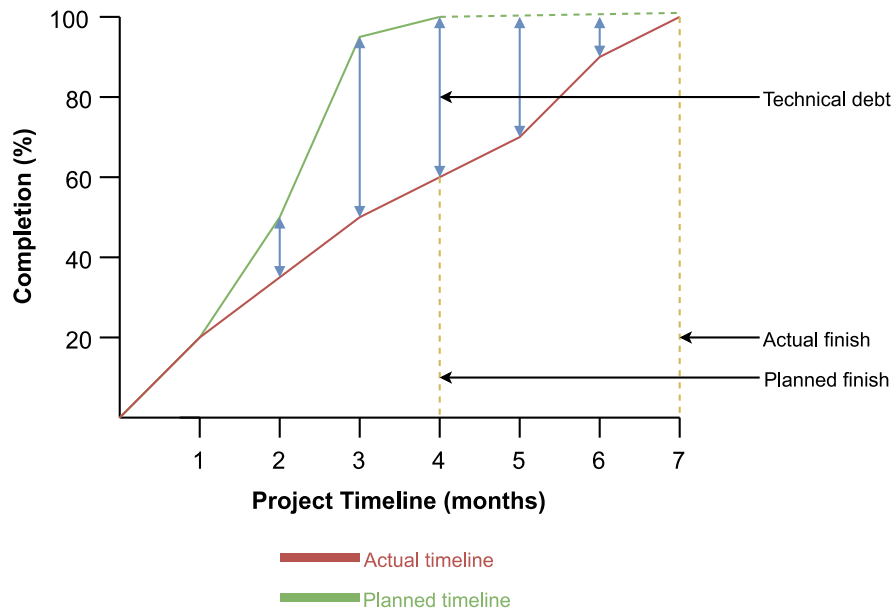


Figure 11. Technical debt visualisation

Fairley and Willshire [27] performed a study on how technical debt may be incurred, managed and mitigated. They found that the following are likely candidates that may cause technical debt:

- During the early stages of a project's life cycle, false assumptions might be made with regards to specifications or features. These assumptions could be proven false at a later stage, which would then require additional time and resources in order to adjust the project requirements in light of the newest information.
- The incursion of technical debt may also be due to a lack of resources, processes, or necessary skills. The loss of personnel with key knowledge or experience with a project, may lead less experienced developers to make uninformed decisions.
- Technical debt may be knowingly incurred if a project's release schedule is under pressure. Certain compromises in quality or functionality may be made in order to meet deadlines.

The consequences of technical debt may be severe and widespread. Software products may be delivered with incomplete functionality, or be of poor quality, with many issues and bugs. A possible lack of documentation could make future development or the use of the product unnecessarily difficult. The most obvious consequence of technical debt may be of a financial nature. Instead of spending

development resources on new development, development time is rather “wasted” on maintaining existing systems and reimplementing features. In the most severe cases, a project may be cancelled due to the high cost of repaying debt, or failure to deliver a specified product on time.

It is also noted that unpaid technical debt may also incur compound interest, meaning that compromises in existing functionality might have a direct effect on future development. If a feature should be expanded, a developer could be forced to work around the limitations of the compromise. Consider the example in Figure 12. When resources are allocated to repay the debt incurred in component A, component B and E (which relies on A) may also be directly affected. The repercussions may further cascade downwards towards components D, F and G as well.

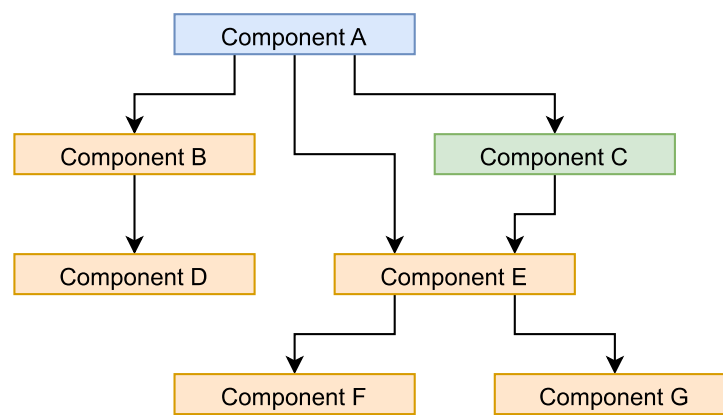


Figure 12. Technical debt dependency hierarchy

In a study done by Izurieta et. al. [28], it was argued that definitions of technical debt are based on issues observed during implementation stages. If technical debt is taken into account during the architectural modelling phases, potential problems which may cause technical debt at a later stage can be identified. Design decisions can then be made early on in the development life cycle to increase the overall software quality.

A technical debt management system was proposed by Vathsavayi and Systa [29] that uses a Genetic Algorithm (GA) to aid organisations in prioritising new development and paying back technical debt. The GA is employed to search for optimal solutions, and the Pareto optimality [30] is used to show the trade-off between focusing on new features and settling the incurred technical debt.

Instead of developing a physical system, Fairley and Willshire [27] proposed several management strategies that may reduce technical debt, or the impact thereof. These include:

- Extend project schedule.
- Change project specifications.
- Add additional resources by hiring more developers.
- Transfer some of the work to subcontractors or consultants.

In addition to the short-term solutions above, the SDM in use during the project's development life cycle may also be adjusted to mitigate or improve the management of technical debt incursion. The waterfall methodology illustrated in Figure 4 on page 12 may be adapted to incorporate technical debt assessments after each phase. The Linear-Predictive Life Cycle (LPLC) is shown in Figure 13.

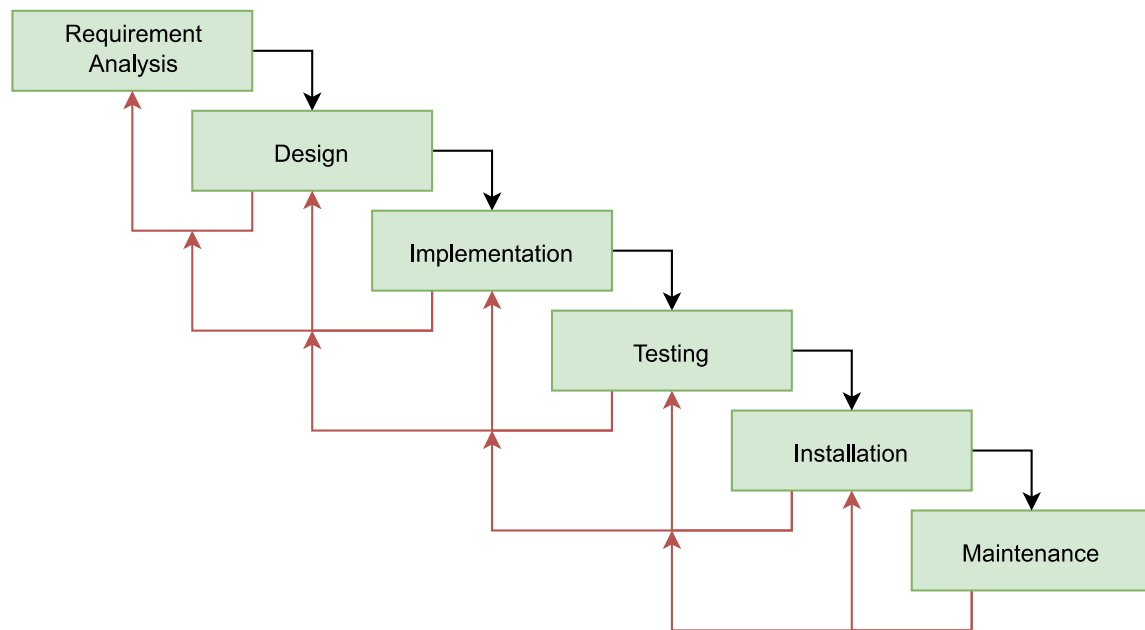


Figure 13. Adapted waterfall SDM

The goal is to end each phase with a milestone review during which potential technical debt is evaluated. Re-planning may occur in order to avoid the accumulation of technical debt during later phases. Alternatively, an iterative SDM similar to an Agile approach may be used to manage technical debt, as shown in Figure 14.

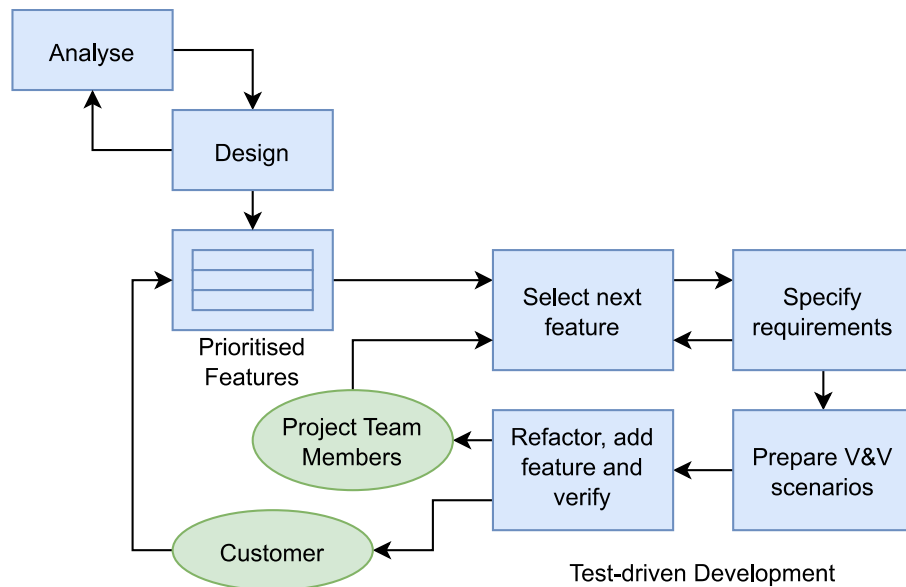


Figure 14. Iterative Adaptive Life-cycle process

Technical debt is managed by constantly iterating through feature development cycles. After a feature is developed, it is validated and verified (V&V) for technical debt incursion. This is directly communicated to the customer, who then may add, remove, revise or reprioritise features. They may, additionally, approve the delivery of a partially completed product.

An alternative method of managing technical debt is to first recognise the fact that some trade-offs will have to be made. Ramasubbu et. al. [31] performed research that suggested the debt load can be tracked by means of three metrics: customer satisfaction, reliability requirements and the probability of technology disruption. These metrics may be mapped on a three-dimensional axis in Figure 15.

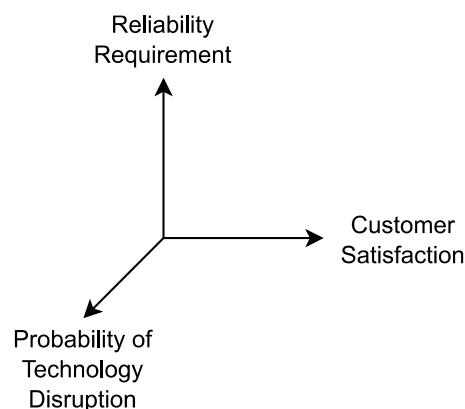


Figure 15. Technical debt metrics

The reliability metric is the level of reliability which is required by the client. If software does not meet the prescribed design standard, the risk of failure is increased. This also increases the technical debt, since time will have to be spent later to ensure that the level of reliability is achieved.

Since software is in most cases designed to be used by an end user, it is important to consider if these end users will be satisfied by the product. Customer satisfaction could be increased in the short term by reducing the product roll-out schedule, or ship the product with additional features. However, care should be taken that these strategies balance the amount of technical debt incurred to avoid long-term dissatisfaction.

Finally, the probability of technology disruption may also impact debt and the payoff thereof. Newer technologies usually aim to improve software development in one form or another. However, it typically takes time for it to stabilise in terms of adoption and integration. A project's debt may be easier to pay off in the short term by using older and more mature technologies. It should, however, be considered that older technologies may contain issues, or lose official development support.

Based on the three metrics above, their research yielded the following findings:

- If both the reliability and satisfaction needs are low, the accumulation of technical debt is of no great concern. All accumulated technical debt may be paid off by switching to a new technology once it becomes available.
- If the reliability requirement is high and the satisfaction need is low, technical debt should be avoided. No functional development should be done until all issues with a new technology have been resolved.
- If the reliability and satisfaction needs are high, technical debt may only be incurred until the product has “taken off”. This is when the product has been adopted by its current user base to the point where they will use it throughout its lifetime. After this, technical debt should be promptly paid off. Payoff may be accelerated if the technological disruption is likely, and the product is switched to the new technology.
- If the reliability requirement is low and the satisfaction needs are high, technical debt may be incurred in order to accelerate product delivery. Technical debt only has to be paid off after the product's growth has saturated. If technological disruption is likely, the product may also be safely switched to the newer technology in order to pay off all accumulated debt.

From the findings above they concluded that a “technical-debt policy must be based both on the business context of a firm and on the technological environment in which the firm operates”. They recommended that software development teams should carefully manage technical debt accumulation and the payoff thereof in light of the three metrics described and shown in Figure 15.

Summary

From this subsection, a number of conclusions can be made in terms of code maintainability and technical debt management:

- Insufficient project requirements, or a sub-standard software design, may lead to low-quality code that decays faster than normal. This decayed code is then also more difficult to maintain.
- Maintenance performed on a piece of software costs time, money and effort during the late stages of a software project's life cycle.
- In most cases, compromises to code quality and maintainability should not be made for the sake of releasing software earlier. Compromised code will likely lead to poor performance of the software product and result in increased maintenance.
- A limited number of maintenance models exists in the literature, and existing models are designed to work only for the specific requirements and environments of organisations.
- Factors that impact maintenance are as follows:
 - Number of developers available to perform maintenance.
 - Is the organisation more focused on rapid application development, or sustainable long-term projects?
 - The development priority from an organisational view in terms of new development versus maintenance.
 - Areas identified in software where maintenance is required.
- There are numerous causes of technical debt, and the time and effort spent in order to repay the debt will cost development resources that could have been spent on new features or products.
- Many strategies exist that may either focus on preventing or limiting the technical debt incurred, or mitigate the effects thereof.
- Software development methodologies may be adapted in order to improve management of technical debt during the software project life cycle.
- Understanding technical debt and the impact thereof may aid in better management of software projects.
- Metrics may be used to analyse technical debt, which may be used to better plan and make compromises based on the information available.

2.4. Software Distribution

In order to ensure that software is executing at peak efficiency, it should be kept up to date. This will ensure that improvements made by the latest iteration of maintenance in terms of bug fixes, security patches, or feature additions are deployed at the client's end. However, the question of how to distribute updates to scattered instances of the software product is of concern. This does not only apply to updates, but to the entire software product as well. An effective distribution model is needed in order for a software product to reach its maximum market penetration.

Stachniak [8] performed a study on the early commercial electronic distribution of software (EDS). During 1970 – 1980 a significant growth in personal computers and video game consoles resulted in a sudden high demand for software. Software was originally distributed in physical packaged form on floppy disk drives, compact disks or cartridges. Additional material, such as documentation, was also bundled in the packaging.

The emergence of EDS services in the 1980s aimed towards reshaping mass-market software distribution. Subscribers to an EDS service could access software by downloading it directly in digital format over a communication network. Some EDS services were free, while others offered paid content.

Today EDS services are the primary form of software distribution. For some platforms, such as mobile phones, an EDS service is the only method by which to install or update software. According to *Statista.com* there are, as of March 2017, over 2.8 million applications available on the Google Play Store, and 2.2 million on the Apple App Store⁵.

There are a few approaches described in the literature in terms of distributing software by electronic means. Dillinger and Becher [11] described various methods of software distribution. Firstly, the client-server model may be used, in which clients send requests directly to the server, as depicted in Figure 16. The client may either request a software update, or the complete software installation package. The client-server model [32][33] is a simple architecture that connects several client nodes to a central server. The server maintains all data and manages communication between clients.

⁵ <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>

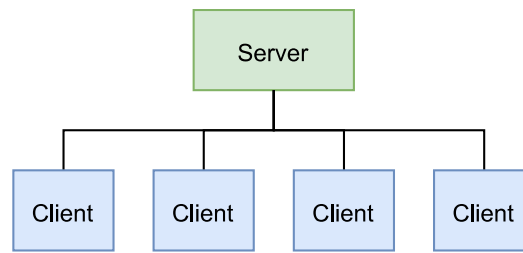


Figure 16. Client-server model

While this is a very simple architecture to implement, it may lead to bottlenecked performance, since the server may be expected to handle requests from multiple clients simultaneously. This scenario is typically expected in a small organisation. An alternative approach may be to use multiple servers in order to balance the request load, as shown in Figure 17.

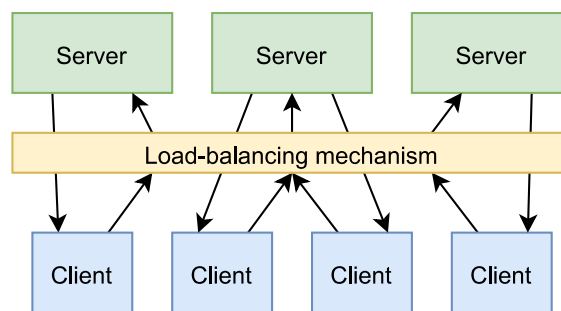


Figure 17. Client-server load balancing

Client requests are directed to a load-balancing mechanism, which then forwards the request to an available server. A decentralised approach is illustrated in Figure 18. A client may initially receive a software update from a server and then act as a server itself by propagating the update to other clients throughout the network.

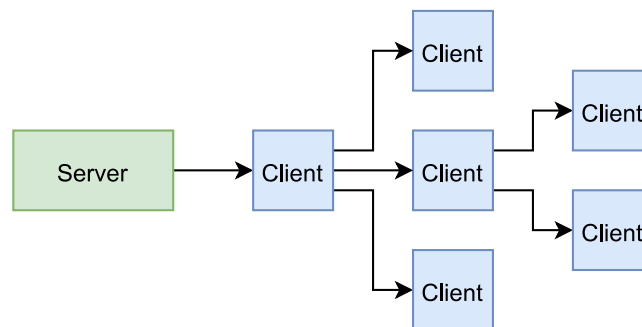


Figure 18. Decentralised distribution model

In an article written by Microsoft, the Windows 10 operating system is described as “Windows as a Service” (WaaS)⁶, which aims to simplify the deployment and servicing of its operating system. Older versions of Windows were released every few years, meaning that the distribution of features and

⁶ <https://docs.microsoft.com/en-us/windows/deployment/update/waas-overview>

security updates were delayed. However, with WaaS Microsoft undertakes to release major updates to its operating system twice a year.

A concept called “service channels” defines specific update programmes, which vary in the frequency by which Windows 10 receives updates. These channels are the Windows Insider Program, semi-annual- and long-term servicing channels, with each aimed towards a specific audience. Firstly, the Windows Insider Program allows expert users to preview new and unreleased features, and provides feedback before Microsoft officially releases an update. The semi-annual channel provides all users with significant feature- and security updates twice a year. Finally, the long-term servicing channel is intended for specialised devices that only require updates every few years.

Several update strategies for Windows 10 are available. The traditional Windows Update utility allows individual computers to receive specific updates. Secondly, the Windows Update for Business Service allows organisations to manage updates for their IT infrastructures. Windows updates may be directly downloaded from Microsoft servers to each computer (by means of the client-server model), or shared across a local network with peer-to-peer connections.

Software applications can mainly be classified into two categories: fat clients and web clients [9][10]. A fat client is software that should be installed on an end device to become operational. Additional software, such as drivers or libraries, may also be required in order for the software application to function correctly. In contrast, web clients are applications that are accessed remotely through a web browser. The web application requires at worst only lightweight dependencies to be installed on the user’s device. However, the software itself typically runs on a remote server accessible to users.

The most significant advantage of a fat client is that it does not rely on internet access in order to function. The application can be run locally on the user’s device, and may or may not require optional internet access for some part to function.

However, fat clients come with a few drawbacks. Firstly, since it requires to be physically installed, it is dependent on the user’s device’s resource in terms of processing power, memory availability, or storage space. The method of software distribution may also potentially present an issue if that specific distribution channel is unavailable to the user. For instance, if a device does not have internet access and the application is solely available online, it can’t be installed. Furthermore, the administrative issues around keeping the software up to date may be cumbersome. If the application is not capable of auto-

updating itself, the user has to perform the task manually. If the application is installed across multiple devices on a network, each instance should be kept up to date, which may cause severe network traffic.

The advantage of using web clients over fat clients is due to the fact that only a single deployment of the software has to exist. This greatly simplifies maintenance, since the application only has to be updated on a single device. All users connecting to the web application will always have access to the most up to date instance. Development is also simplified due to the fact that the application only has to execute within a single operating environment. A user's device does not need to have all the dependencies installed in order to run the application. Since the application is web based, the only requirement is that it should be able to run on a range of web browsers.

Web applications are, however, not without drawbacks. Due to the nature of being web based, an active internet connection is required. Some web applications may also require a large amount of bandwidth in terms of either capacity or speed. If a user's device does not have internet access, the application cannot be accessed. Finally, the standard for web-based programming languages is constantly evolving. This requires that the application should remain backwards compatible with older technologies or browsers in order to maximise compatibility.

In recent times, the distribution of software has shifted towards a Software-as-a-Service (SaaS) approach. SaaS is a distribution model where providers offer web-based applications to consumers [12]. Consumers will typically pay a subscription fee in order to use the SaaS product. By using SaaS, a consumer will be able to separate the management and operation of its software (or certain parts thereof) [34]. The management aspect will remain with the consumer, but the operations in terms of hardware, networking and support become the provider's responsibility.

The cloud-based approach of SaaS allows organisations to implement applications with requirements in terms of mass data processing, time delay sensitivity and reliable data transmission [32], without investing considerable time, money and effort into development. The organisation essentially rents the use of third-party infrastructure for processing and data storage.

There are currently many examples of popular SaaS applications. First and foremost is the range of Google products⁷, aside from its search engine. Table 1 lists a selection of Google products. In addition

⁷ <https://www.google.com/intl/en/about/products/>

to Google products, other organisations also provide a wide variety of cloud-based services, such as Dropbox⁸, Amazon Web Services⁹, Microsoft Azure¹⁰, etc.

Product Name	Description
Google Drive	A cloud storage service which can sync files across multiple devices.
Gmail	An online email application, which can also sync and integrate with existing email clients.
Google Play Music	An online music streaming service.
YouTube	An online video service.
Google+	A social networking tool.
Photos	An online image storage service which can sync photos across multiple devices.
Calendar	An online calendar application which can sync meetings and events across multiple devices.
Keep	A note-keeping service.
Docs	An online document editing application.

Table 1. Selection of Google products

Summary

The following list summarises the findings and conclusions from this subsection:

- An effective software distribution model is necessary to ensure that a software product or update maximises market penetration.
- Various distribution models exist, each suited to its own environment.
- Software may be categorised as either fat clients (desktop software) or web clients.
- With SaaS gaining popularity, more applications are being offered as online solutions.
- Since web applications only require a single deployed instance from which users may gain access, the issue of keeping fat-client software up to date through a suitable distribution model is eliminated.

⁸ <https://www.dropbox.com/>

⁹ <https://aws.amazon.com/>

¹⁰ <http://azure.microsoft.com/>

2.5. Web Development

Section 2.4 established the fact that distributing software online as web applications simplifies the development and maintenance process. This subsection will discuss various web-based frameworks, as well as implementations thereof, from the literature.

A web framework can be described as a set of tools and libraries to aid in the development of web applications [35] [36]. These tools and libraries present developers with reusable components that may be used to speed up development. A framework for rapidly developing mobile applications was proposed by Mnaouer et. al. [37]. The purpose of the mobile applications is to dynamically display interfaces based on certain workflow information. Their framework uses existing architectures and open source standards, such as eXtensible Markup Language (XML), Simple Object Access Protocol (SOAP) and Web Service Definition Language (WSDL).

The framework is service-orientated, meaning that resources may be accessed through XML-based protocols such as SOAP. The interfaces and bindings to which clients connect may be defined using WSDL files. Each service deployed on the framework is a component in itself, which may either be used in isolation, or in combination with other service components. The framework therefore consists of a set of generic web services which may be reused for various applications.

The proposed model for their framework is based on the Web Service Reference Architecture (WSRA) framework authored by SUN Microsystems and Infocomm Development Authority. The main requirements of the framework are to provide the following:

- A login service to provide users access to the web services. It allows the creation of accounts and login authentication.
- A core data source service which allows other web services to connect to various types of data sources.
- A workflow update service from which mobile applications may download workflow files.

In the industrial sector, Fleischmann et. al. [38] proposed a modular web framework for Condition Monitoring Systems (CMS). CMS monitors data received from various hardware sensors and provides fault predictions and tolerance. Implementing CMS is challenging due to the distributed nature of computational tasks from various “Internet of Things and Services” (IoTS) architectures. The proposed framework aims to support the maintenance process by offering detailed condition-monitoring functionality.

The framework should provide a central point of access to users, and provide them with near real-time data. Its user interfaces should also be adaptable to different kinds of visualisations. Finally, the framework should be flexible in terms of data analysis and machine learning techniques. To accomplish these requirements, the following standards and technologies were used:

- Hypertext Markup Language (HTML) 5
- JavaScript (JS)
- Node.js
- Open Platform Communications Unified Architecture (OPC UA)

Node.js is a cross-platform solution developed by Google for web applications. It is an event-driven architecture for real-time applications and offers a high level of scalability. HTML 5 and JS are used to implement the client-side application that connects to the CMS server.

A web-based management system for Wireless Sensor Network (WSN) monitoring was designed by Naruephiphat et. al. [39]. The system was initially implemented for two WSN applications: power monitoring and temperature/humidity monitoring, but was ultimately designed to easily support additional WSN applications in areas such as environmental monitoring, healthcare or transportation. The Model-View-Controller (MVC) architecture was used in the design of the system.

There is a need for web-based solutions in the medical field as well. Long-term care provided to patients includes health, social, personal and supportive care [40]. Worrall and Chaussalet proposed a web-based solution to assist medical professionals to make decisions based on available data and observations made. The system should provide analysis and reports on historical medical activities for a patient. Reports should be generated, which must then easily integrate into existing documentation. The system should also take into account that non-technical users will operate the system. The system was finally developed using the MVC architecture.

MVC is a design pattern that is used to separate application logic from the user interface [39] [40] [41] and is shown in Figure 19. The *Model* is generally a class or object that represents data stored in a data source. The *View* defines the Graphical User Interface (GUI) presented to the user and is responsible for displaying models. The *Controller* handles the core application logic, such as receiving and processing user input and then returning a result to the user.

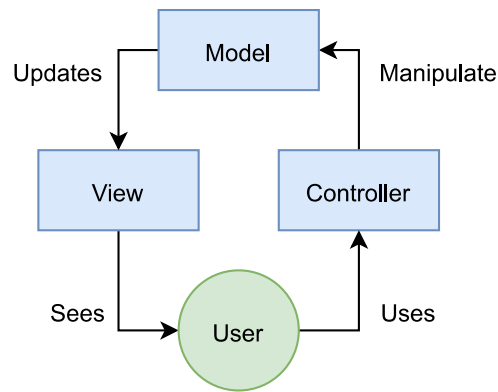


Figure 19. MVC architecture

These three main components are independent of each other, meaning that it is easier to, for example, make changes to the user interface without changing any of the processing functionality, or the models that describe the information displayed to the user. This advantage also makes it simpler for multiple developers to work concurrently on a single application.

Since MVC is an architecture, it is not specific to a programming language. It has been implemented in the following languages in the literature:

- ASP.NET [36]
- Java Server Pages (JSP) [42]:
- PHP [43]
- Python [44]

In a study conducted by Jaila et. al. [41], two languages were used to implement MVC and compared to each other in terms of performance. The performance metrics used were page load time, request transfer speed, response transfer speed, etc. Their findings, as shown in Table 2, concluded that the ASP.NET language performed better according to every metric.

Profile	ASP.NET	PHP
Page load time (ms)	370	676
Time to first byte (ms)	213	383
Time to last byte (ms)	165.69	293.47
Request transfer speed (kbps)	10.68	5.938
Response transfer speed (mbps)	0.612	0.101

Table 2. ASP.NET vs PHP performance evaluation

Summary

To summarise, this subsection described a web framework as a collection of tools and libraries to assist developers with rapid application development. Various frameworks with a wide variety of technologies were implemented, including HTML 5, JS, ASP.NET and PHP. MVC is considered a popular architecture for designing web frameworks.

2.6. Summary of the Literature

By examining software development life cycles, it is evident that a challenge exists to produce high-quality software in a reasonable amount of time. By following a good methodology, technical debt can be reduced to minimise the time and effort spent on software maintenance.

In order for a software product to reach its optimal market penetration, an effective distribution model is required. Software updates and maintenance is greatly simplified by distributing software online, or in the form of software-as-a-service. A web framework provides developers with tools and libraries to shorten the development timeframe, and effectively reuse existing functionality.

Key technologies and languages identified in the literature are used to design and implement a solution in chapter 3. These include web languages such as HTML, JS and ASP.NET, and the MVC architecture. MVC as an architecture provides the means by which to create a modular environment. Individual components each has a specific role in the overall architecture, and may interact with various other components.

Chapter 3

Design of a Modular Framework

3.1. Preamble

This chapter will focus on the design and development process of the framework. The term “Framework” is hereafter used to describe the system that was implemented to solve the problem described in section 1.4. This reference will include all aspects discussed in section 3.4 and 3.5, including best-practice guidelines for developing a web application on the framework.

The term “Platform” is used to describe the entire solution: the framework and all web applications that have been developed on the framework. The platform is therefore the end-product that the user has access to. The relation between the framework, platform and web applications is illustrated in Figure 26, section 3.4.3.

This chapter will define the requirements for the framework, and provide a detailed design of the framework that was developed. Finally, it is verified that the developed framework met the required specifications.

3.2. Framework Requirements

Before the development of the framework could commence, some design specifications were set. These were identified from the need for the study section in chapter 1.3, as well as from requirements set by the ESCo.

3.2.1. Database

In order for the framework to be compatible with the ESCo’s existing database system, MySQL should be used. This will allow access to all the ESCo’s current and historic data, such as user account information, energy measurement data, etc. At the time of writing, the database contained 191 tables and an estimated total of 131 million records, growing daily.

3.2.2. User Authentication

Existing user accounts stored in the database should be authenticated. This will eliminate the need for the ESCo’s users to have multiple user accounts, and maintenance of these accounts will then apply to both new and existing systems. The pseudo-code below describes the authentication procedure in existing systems.

```

Set authenticated to false
Query database for users with access rights and an enabled password
For each user
    Decrypt password used for login using the AES256 method
    Validate decrypted password by hashing the decrypted password with a unique
    salt value
    If password is not valid
        Continue with next user
    Otherwise
        Set authenticated to true
        Query all user data from database
If authenticated
    Save user data in session
    Return true
Return false

```

This authentication process may be reused by various components.

3.2.3. Accessibility

In order for the platform to be centralised, cross-platform compatible and accessible from the web, a web-based development approach is required. By distributing the platform from a web perspective, it is ensured that all users will have access to the latest version of the platform. Figure 20 provides a graphic representation of how various users on different devices and operating systems connect to the platform.

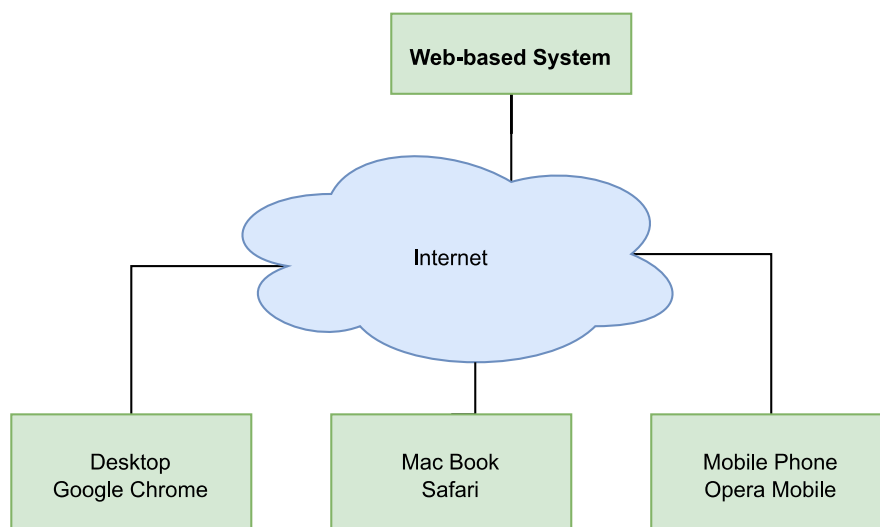


Figure 20. Multi-platform web-based user access.

A user's web browser, regardless of the type of device it is running on, such as a desktop computer, laptop or mobile phone, is able to access the platform through the internet at its web address.

3.2.4. Access Control

In addition to authenticating existing user accounts from the database, additional access control is required. Access control determines to which resources and applications a user have access to within the platform. It is required of the framework to implement a mechanism that assigns specific privileges and access rights to user accounts, which prevents them from accessing unauthorised areas.

3.3. Development Software and Systems

The framework was developed using the environment, languages and dependencies described in the subsections that follow.

3.3.1. Development Environment

This subsection briefly describes the software and tools used in the environment in which the framework was developed.

Visual Studio

Microsoft Visual Studio 2017 Community Edition¹¹ was chosen as the Integrated Development Environment (IDE) with which the framework was developed. It is a free IDE that supports a wide range of development languages and frameworks, such as C#, ASP.NET and MVC.

ReSharper

*ReSharper*¹² is a product from *JetBrains* that provides extensive add-on tools for Visual Studio, including code analysis, refactoring, navigation, code formatting and code generation. These tools are ultimately used to keep code in a good condition to minimise technical debt during development. ReSharper may also be configured with custom coding styles and will notify the developer if a piece of code does not follow the standard.

¹¹ <https://www.visualstudio.com/>

¹² <https://www.jetbrains.com/resharper/>

Internet Information Server

Internet Information Server (IIS) 8.5 is included in Windows Server 2012, which is installed on the server where the platform is deployed. IIS is used since it manages domain names and redirects requests for a web service to the corresponding web application.

3.3.2. Languages and Dependencies

The framework uses the following selection of programming languages and dependencies. The jQuery and Bootstrap dependencies are third-party libraries which provide additional client-side functionality, as well as visual components. Additional libraries and tools may be added to the framework via the NuGet Package Manager (NPM). NPM maintains an online directory of open-source libraries, which may be added freely to development projects.

ASP.NET

Microsoft ASP.NET¹³ is an open-source web language for creating web applications using .NET Framework libraries. Extensive libraries and resources are available. ASP.NET was chosen as the primary programming language, since it provides a large collection of libraries, supports MVC applications, and is extensively documented.

jQuery

The jQuery website¹⁴ describes it as a “fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.”

The framework uses jQuery to provide all client-side functionality for web pages. Ajax is extensively used to provide dynamic communication between the client and server for content updates.

Bootstrap

Bootstrap¹⁵ is an open-source front-end framework that allows developers to build responsive, mobile-first user interfaces. It uses a combination of JavaScript (JS), Cascading Style Sheets (CSS) and HTML to provide a library of visual component templates.

¹³ <https://www.asp.net/>

¹⁴ <https://jquery.com/>

¹⁵ <http://getbootstrap.com/>

The framework uses Bootstrap to style many visual elements and ensures that web pages are rendered correctly on both desktop and mobile environments.

3.4. Technical Design

This subsection will discuss the framework and all its components in detail. In order to provide insight into the architecture of the framework, the principles of MVC will be briefly discussed. Diagrams will be given to provide both high- and low-level overviews of certain components, and interaction between the various components will also be discussed.

3.4.1. MVC Architecture

As discussed in chapter 2, MVC is a popular web architecture which is used to separate application logic from the user interface.

The **Model** is a representation of information stored in a data source. There is typically a direct correlation to the model's properties and the columns found in a data table. For instance, if a table called *Students* in a database is represented by Table 3, the Model class representing this table will look as follows in C#:

```
public class Students
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Students	
Column name	Data type
ID	Auto-number
Name	Text
Age	Number

Table 3. Example database table

The **View** is the GUI presented to the user. In order to display information contained in a Model, that Model should be bound to the View. When bound, the View has full access to all the Model's properties, and may display them however is required. Figure 21 shows the binding between a View and a Model. The View is presented to the user's web browser as an HTML document.

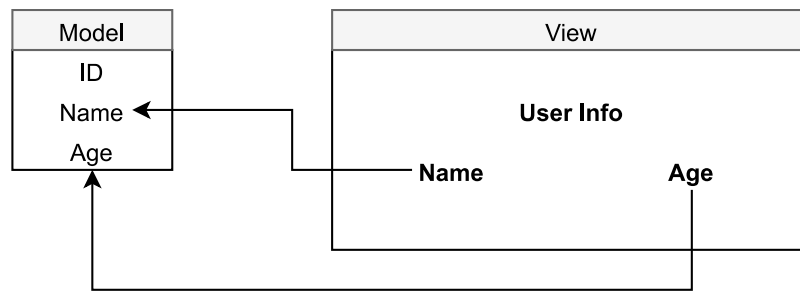


Figure 21. Model binding

The **Controller** handles all the application logic, processing and data manipulation, and listens for client requests. When a request is received by the Controller, it responds by returning a View after the necessary processing has been performed. A Controller is divided into separate *actions* or functions, where each action listens for a specific client request from a user's web browser. A Uniform Resource Locator (URL) is mapped to each Controller action and is referred to as "routing" in MVC (discussed in section 3.5.2).

Figure 22 illustrates the relationship between the Model, View and Controller.

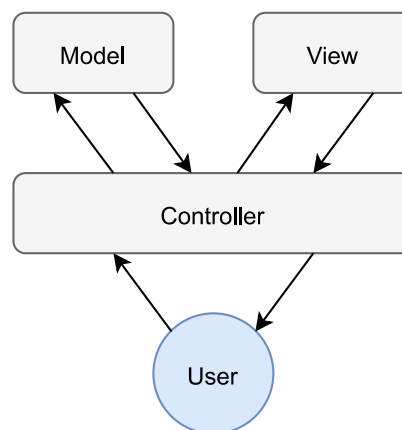


Figure 22. MVC process overview

The user initiates the process by sending a request to the web application for a specific page. The Controller accepts the request and retrieves the necessary Model. It then passes the Model to the View in order to construct the web page. The web page is finally returned to the user's browser.

3.4.2. Extension on MVC

For the implementation of the proposed platform, the standard MVC architecture was extended to include two more concepts: a Data Access Layer (DAL) and View Model (VM). These aim to further abstract common functionality into separate and maintainable areas. Figure 23 illustrates the process, including the extensions for MVC during a client request.

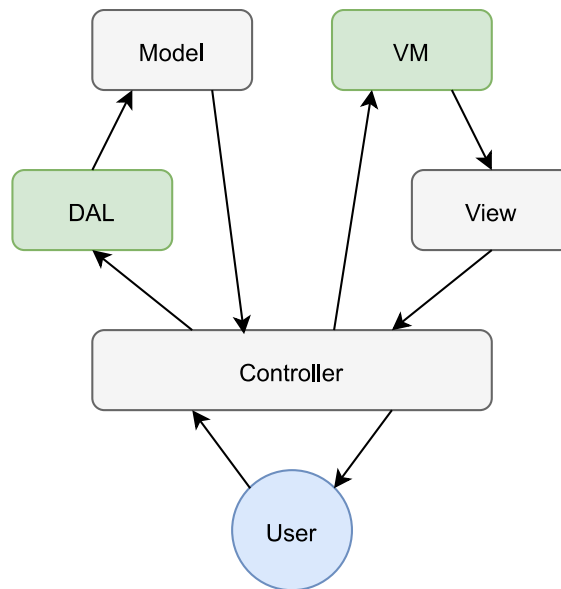


Figure 23. Extended MVC process overview

Similar to the standard MVC process, a user will send a request for a specific web page, which is routed to a Controller's action. In this case, the DAL retrieves data from a specific source, such as a database, and converts it to a Model. The Model is packaged into a VM, which is then bound to a View. Finally, the HTML document generated from the View is returned to the user's web browser. The DAL and VM is explained in more detail below.

Data Access Layer

The purpose of the DAL is to centralise all database-related operations into a single area. By grouping database transactions together into a single class, maintenance is simplified, since the developer knows exactly where to find a specific database query function.

A single DAL class for every table in the database schema was added to the framework. This is needed since each table requires basic Create, Read, Update and Delete (CRUD) operations, including any specialised database queries. If all queries for all tables were provided in a single class, that class would simply become too large to maintain properly. Figure 24 shows a conceptual example of where the DAL fits in.

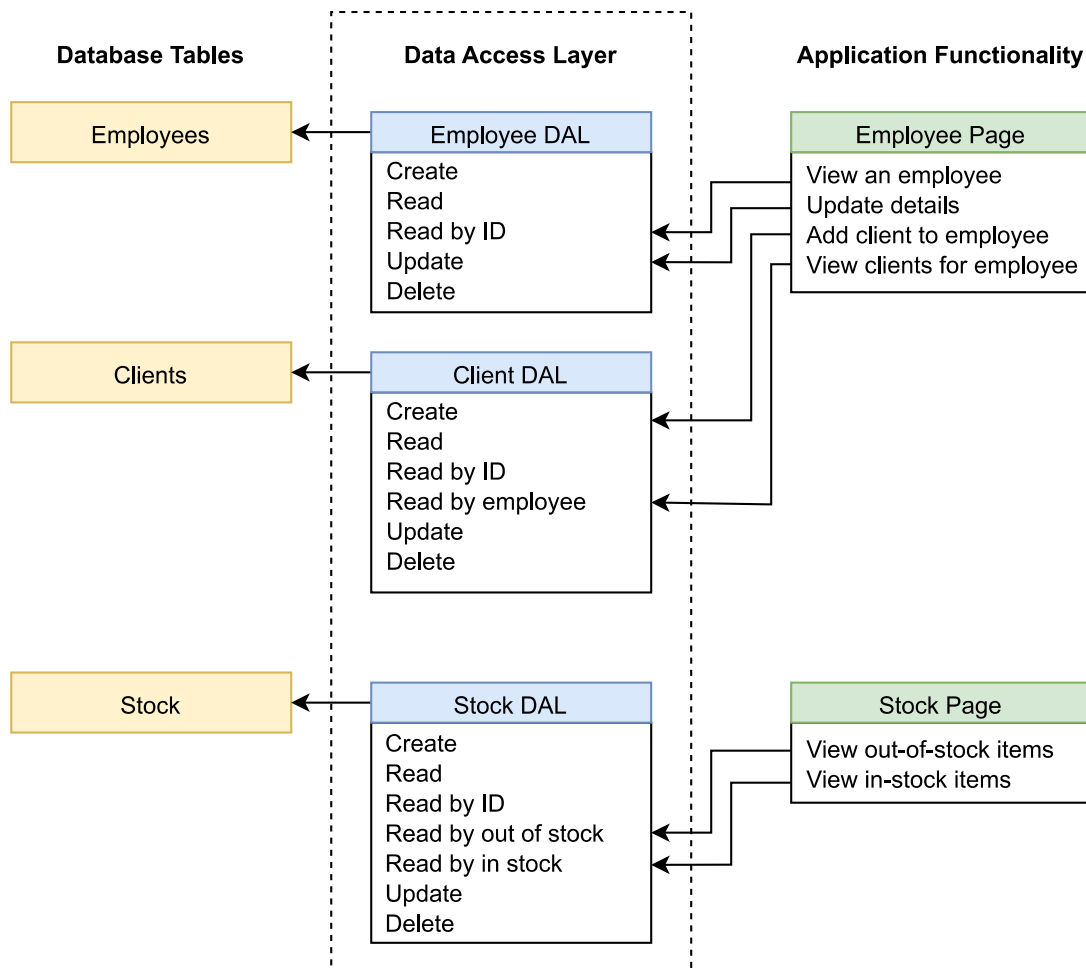


Figure 24. Data Access Layer

In this example, a DAL exists for each of the three tables. Each DAL contains the basic CRUD operations, including custom queries, as required by the application logic functionality. When information from a specific table is required, the corresponding DAL is queried.

View Model

A VM is simply a container for one or more Models and/or miscellaneous properties. Since only one “model” may be bound to a View, the use of a VM allows multiple Models and their properties to be accessed from within a View. By looking at a VM class, it can also easily be determined what kind of information is displayed in a View, thereby improving maintainability. Consider the example in Figure 25:

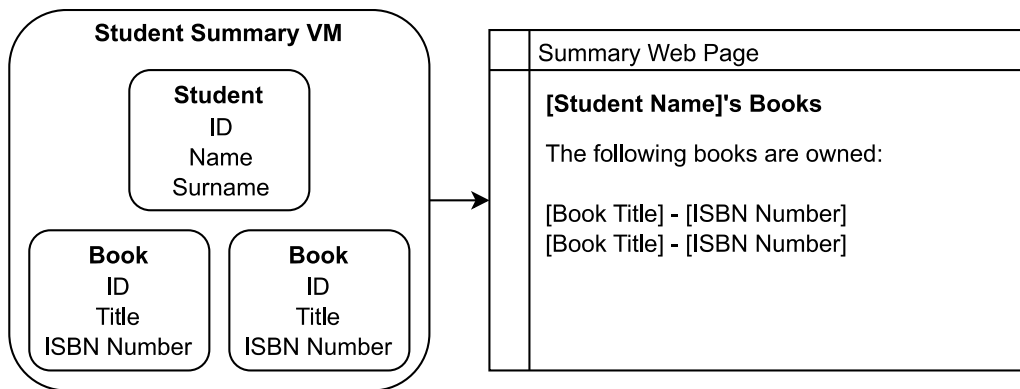


Figure 25. VM example

If it is required to display a list of books owned by a student on a page, the VM would include Models for the student, as well as a collection of Books Models. The VM is bound to the View, which displays the student's *Name* property as the page title, and both book *Titles* and *ISBN* numbers.

3.4.3. Framework Overview

This section will provide an overview of the developed framework, and how the various components fit together. The framework provides an architecture on which web applications can be rapidly built by using a collection of available tools and libraries. These tools and libraries act as “building blocks”, where the framework itself is the foundation, and web applications may be constructed by assembling the required functional blocks. Each web application on the platform is developed by following a best-practice guideline, which ensures code consistency to make maintenance easier and prevent future technical debt.

As shown in Figure 26, each web application is contained within its own Area. In ASP.NET, an Area is described by Microsoft¹⁶ as “an ASP.NET MVC feature used to organize related functionality into a group as a separate namespace (for routing) and folder structure (for views)” and “Areas provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is effectively an MVC structure inside an application.”

¹⁶ <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas>

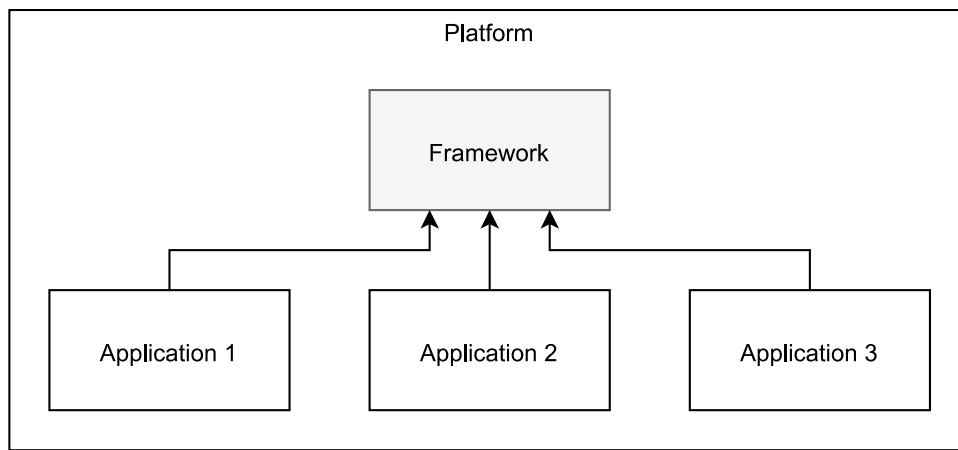


Figure 26. Platform structure overview

By using an MVC namespace structure, applications can easily be derived from the main framework, which saves time. In software development, a *namespace* is a term that refers to an object hierarchy to group similar objects. It also allows the use of names in different contexts and is achieved through the use of a folder structure, where each folder represents a single namespace. This folder structure is the hierarchy of folders located in the Visual Studio project, and each folder contains source code files.

Consider the example in Figure 27. A total of seven namespaces exists, one for each folder present. Both Application 1 and 2 are allowed to have a “Content” and “Scripts” folder, since they are located in separate Areas with different namespaces/folders.

The framework consists of many components in addition to those that form part of the MVC architecture. Figure 28 illustrates a detailed overview of the framework components. Each component may be categorised as either GUI, data, client-side content, assets, or functionality – although these categories are not directly related to any namespaces.

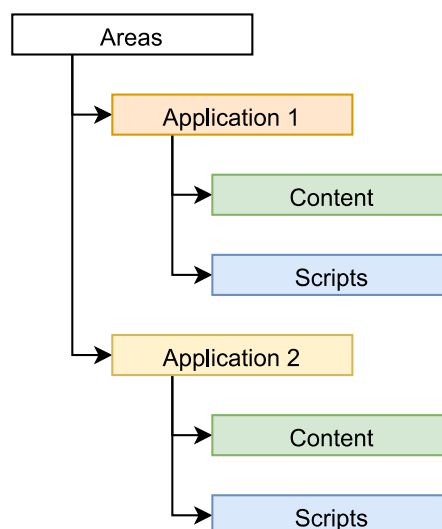


Figure 27. Area and namespace hierarchy

The data category contains components that relate to data operations and containers; the DAL for the former and Models/View Models the latter (as discussed in sections 3.4.1 and 3.4.2). Client-side content contains all resources which will be requested by the client browser, such as the web page's CSS and JS references. Each JS component is discussed in more detail in Appendix 1. Assets contain all image-related resources of the platform. Libraries and filters contained in the functionality category are common functional components that are available to the entire platform and are discussed in further detail in Appendix 1.

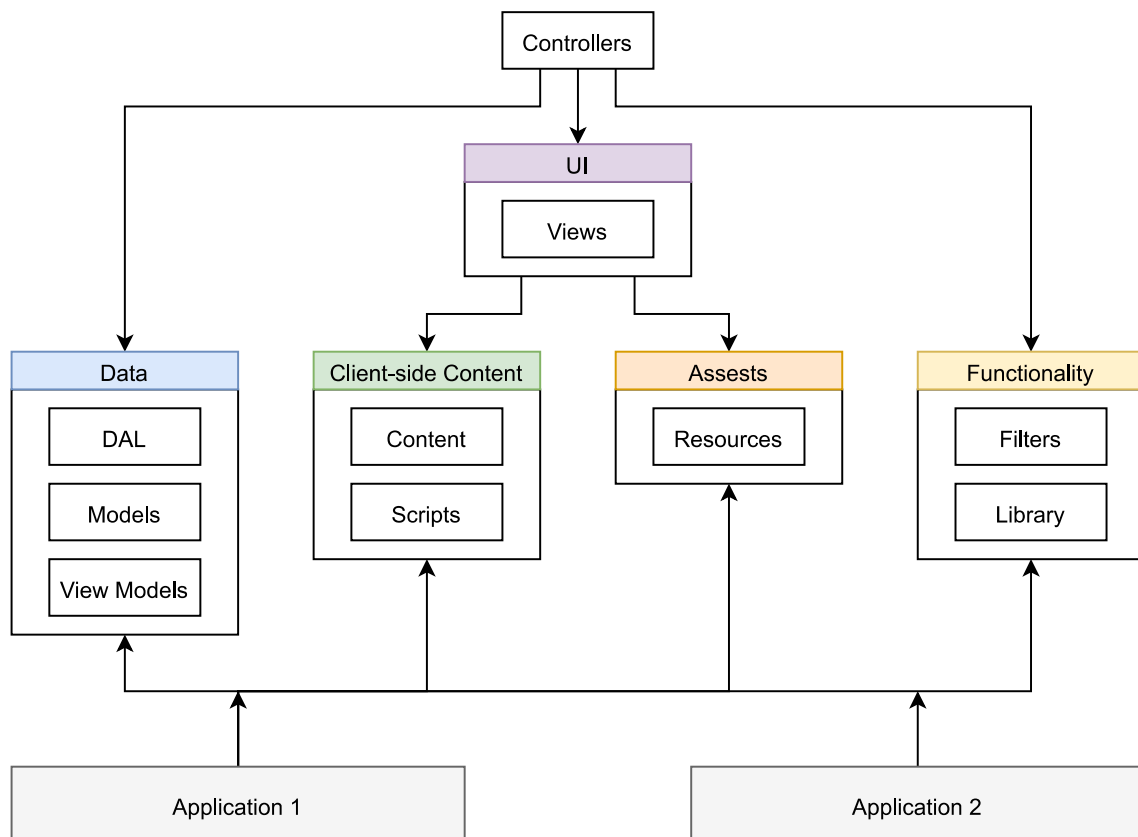


Figure 28. Framework structure overview

All components in the categories mentioned above and shown in Figure 28, except the Controllers, are accessible to all web applications on the platform. These Controllers are responsible for framework-specific actions and are discussed in more detail in section 3.4.6. As shown in Figure 29, each application has a component structure similar to that of the framework.

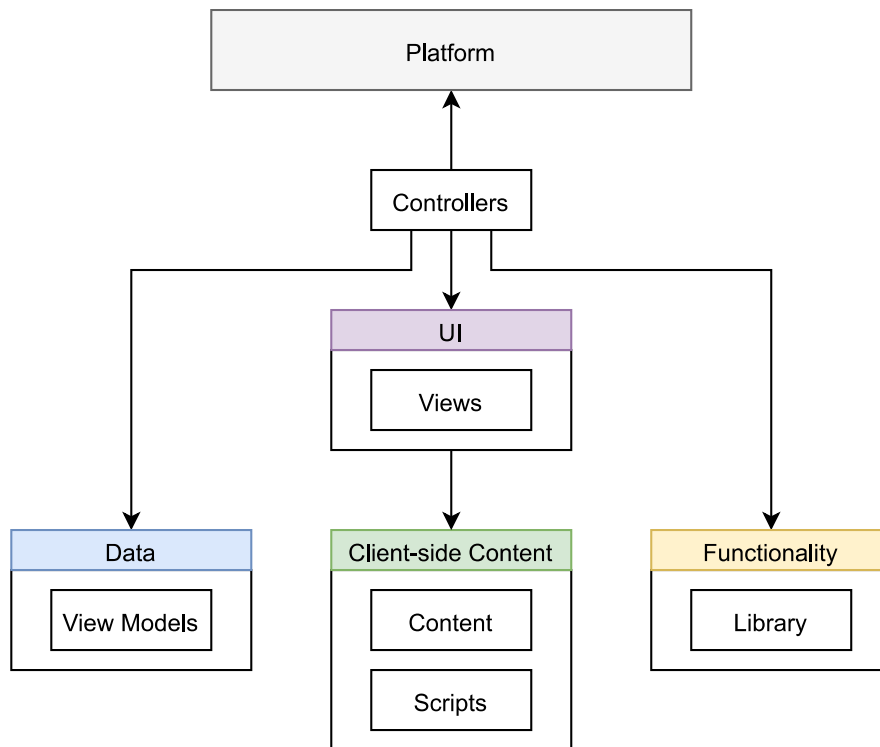


Figure 29. Application structure overview

Since DALs and Models are considered to be *per-database-table* instead of *per-application*, they do not form part of an application's data category. View Models are created as data container for specific Views and are therefore included in the application's structure. Since the client-side content is also directly related to a View, it also forms part of the application's structure. Assets and filters are made available to the entire platform and are also omitted from the application structure.

3.4.4. Models

The framework Models are divided into two separate namespaces, namely *Database* and *Custom*. The *Database* namespace contains Models that map directly to database tables, as is the convention for MVC Models, as discussed in section 3.4.1. *Custom* Models are used for DAL queries that return Structured Query Language (SQL) results that can't be mapped directly to a Model, such as a JOIN¹⁷ query between two tables.

3.4.5. Data Access

A base DAL class was designed to contain all the necessary functionality in order to connect to a database, and read and write data from/to a table. By using Object Orientated Programming's (OOP) inheritance concept, any table's DAL class may be derived from the base DAL class and inherits all its

¹⁷ A MySQL JOIN operator is used to combine result sets from two or more tables, based on a column that exists in both tables. For more details, see https://www.w3schools.com/sql/sql_join.asp

functionality. This has a significant impact on maintenance, since changes or additions to the base class automatically reflect to all its heirs. Custom read and write functions are then implemented in each derived DAL. Figure 30 illustrates an example of DAL inheritance.

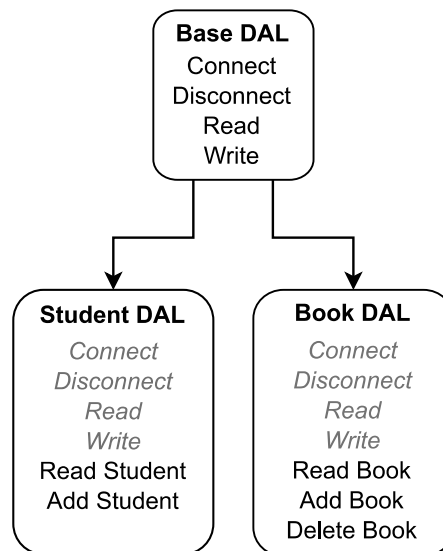


Figure 30. DAL Inheritance

The base DAL contains functions to connect and disconnect from a database and provides basic interfaces for reading and writing to a table. The Student DAL is used to manage students registered in a *Student* table. Since it inherits from the base DAL, it has access to all its functions. Two additional functions are provided to read Student entries and add a new Student entry to the table. The same holds true for the Book DAL, but it has three other functions of its own.

Similar to the approach taken on namespaces for Models in section 3.4.4, DALs should also be placed in either the *Database* or *Custom* namespaces for DALs. A *Database* DAL should only contain read and write functionality that applies to the associated database table, where a *Custom* DAL may contain SQL transactions for queries that cannot be directly associated with a single database table.

3.4.6. Controllers

The framework contains a few exclusive Controllers that provide basic GUI functionality. These Controllers are used for:

- Login
- Dashboard
- Errors

The login controller provides users with an interface from which to sign into the platform. After a successful authentication process (see section 3.5.1), the user is redirected to the Index¹⁸ action of the dashboard Controller. The dashboard is essentially a landing page for the framework, which presents the user with a selection of web applications that are implemented on the framework. The error Controller is used to handle all platform errors. Although this Controller is not directly accessible to web applications on the platform, the ASP.NET framework will automatically invoke this Controller when errors occur. See section 3.5.3 for more details on how error handling is performed on the framework.

3.4.7. User Interface

The web page that is presented to the user is constructed from various elements, such as the HTML structure and CSS styling.

View Layout

In ASP.NET, a layout defines a common structure for use between Views and ensures a consistent user interface for a web application. The layout should typically define all header information, such as references to CSS and JS, including the base HTML structure, such as navigational menus, headers and footer bars.

Individual Views then references the layout to first include the base HTML structure, and then add additional content as required. In some cases, certain information that is displayed in the layout should differ between Views. If such is the case, a *section* may be defined within the layout. Each View may then specify the content that should be rendered within the *section*. Such information may include a page title, logged-in user's name, current time, or optional content in specific areas of the page.

An example is shown in Figure 31, where two Views are derived from a layout. Both Views share a consistent design, but with their own page title and contents.

¹⁸ An *Index* action is the default action for any given Controller. If the action name is omitted from the URL, the Index action will be called by default. See section 3.5.2 for more details on URL routing.

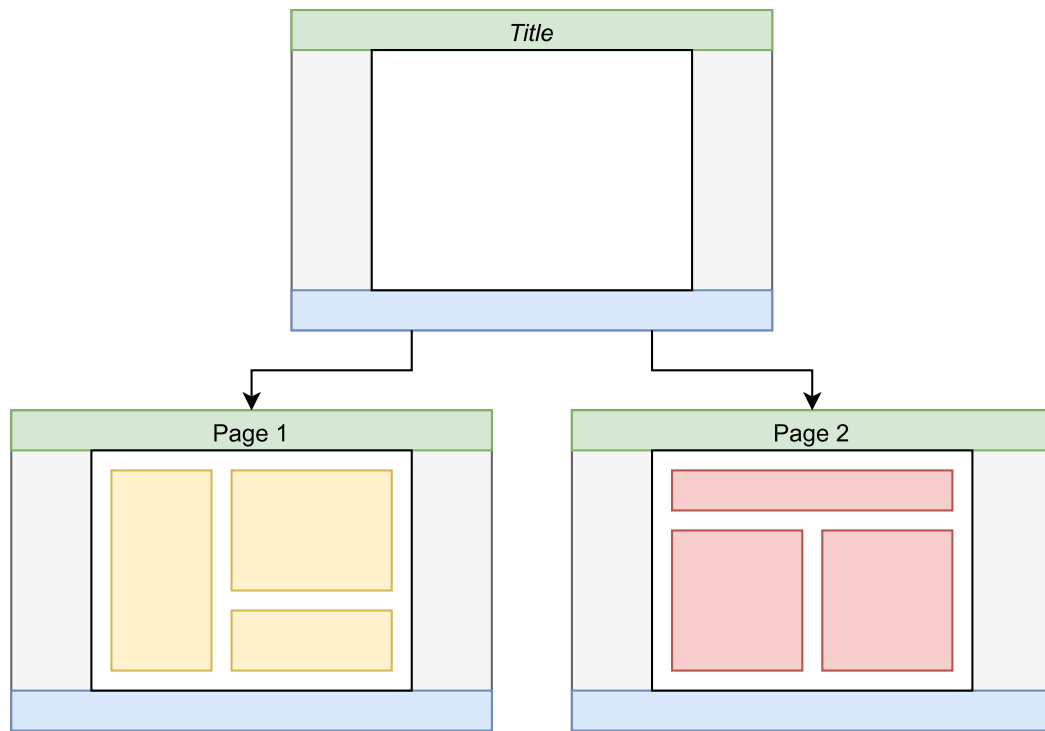


Figure 31. Layout inheritance

The layout used by the framework is illustrated in Figure 32, and aside from the HTML structure consists of three content *sections*. These *sections* are page title, scripts and sidebar. The scripts *section* defines any additional CSS or JS references that should be added to the page, and the sidebar *section* is a collapsible area to the left side of the screen.

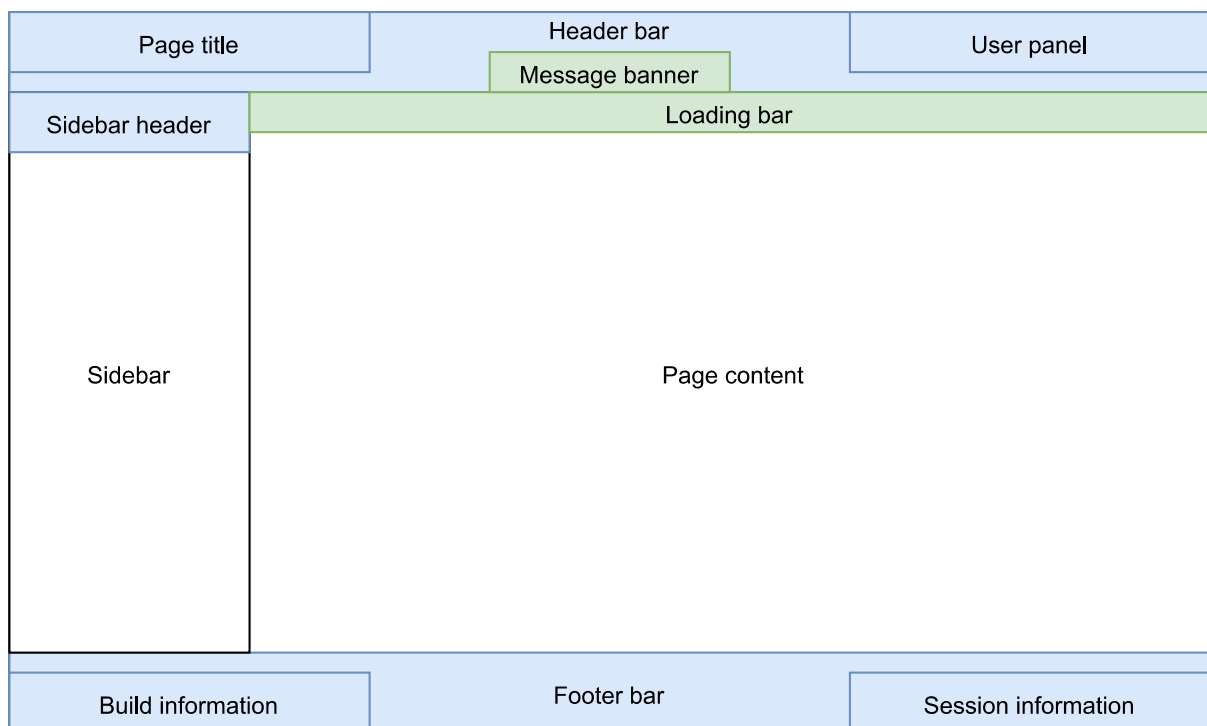


Figure 32. General page layout

The header bar contains several components:

- Page title that may be set by the inheriting View
- User panel that displays information of the current logged-in user
- Message banner that may be used by inheriting Views to display notifications to the user

The sidebar *section* may optionally be used by pages to display any necessary content, such as data filters, page options, or navigational items. If the sidebar is enabled for a page, the user may still choose to dynamically hide or show it by selecting a collapse button located in the side bar's header.

When a View is created and all sections have been defined, all remaining content is then rendered in the page content area. This area is scrollable between the header and footer bar. A loading bar is available that may be toggled on or off to provide visual feedback when a script is busy with a server request.

The footer bar contains an area where build information, such as the build date and version number, is displayed. Another section contains session information, indicating to the user the time left before his session will expire. The session may be kept alive by any user interaction, such as moving the mouse pointer, or entering data into a field.

Style Sheets

The framework uses a global CSS that is referenced in the layout View. This CSS defines the colour scheme and finalises the page structure by placing certain elements in their correct positions relative to others. When a page is derived from the layout, as shown in Figure 31, that page should reference an additional CSS file for any other visual elements that are unique to that page. These CSS files are placed in the *Contents* folder relative to their area.

3.5. Additional Features

In addition to the architecture and design discussed in section 3.4, this subsection provides details on a selection of additional features of the framework.

3.5.1. Platform Security

Some form of access control is required, since it is not desired to provide free access to any user on the internet to all platform resources. This would require a user to provide credentials, which should be

authenticated before allowing the user access to the platform. Even if a user may access the platform itself, it does not necessarily mean that the user may access all the platform's resources. Additional access control is then required to ensure that the user may only view certain web pages.

Login and Authentication

When users access the platform, they are first presented with a login page that is returned by the *Login* Controller. When a user submits their login credentials, the *Authenticate* action in the Controller checks if the login credentials are valid. The complete authentication process is illustrated in Figure 33.

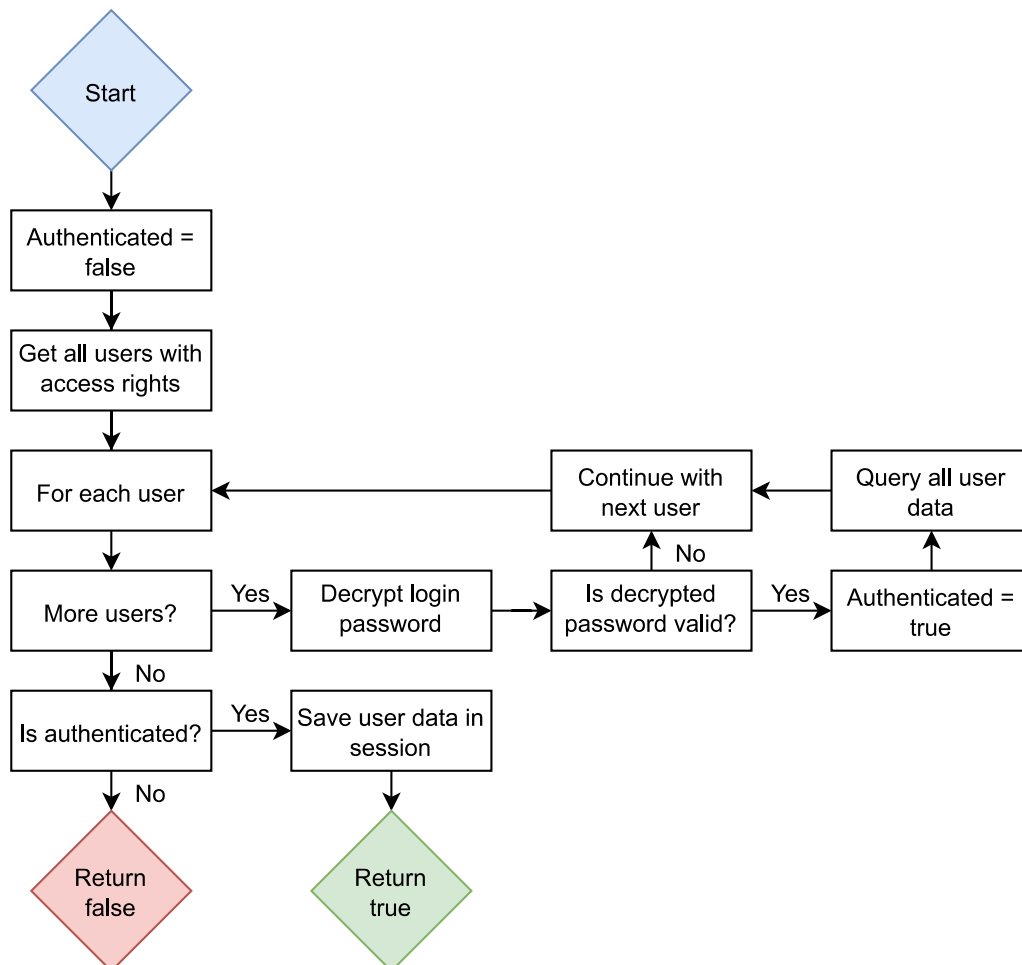


Figure 33. Authentication process

Once authenticated, a session is initiated that contains information about the user, such as the user's name, surname, email and user rights (discussed below). The user request is then redirected to the index page of the *Dashboard* Controller, where a selection of productivity tools are available.

If an independent external system needs access to a resource, or triggers an action, authentication should also be performed. In this case, the *Controller* action that is requested should be secured with the external login filter, as discussed in Appendix 1. This filter performs the same process shown in

Figure 33 and will execute the action if the request has been authenticated. If invalid credentials are passed to the action, the filter will return an HTTP 401 (forbidden) error response.

In order to ensure that inactive user accounts do not maintain access to the platform, a security feature was added that periodically scans through all user accounts. Accounts which have not logged into the platform for a period of ninety days are then disabled. This feature ensures that old accounts from former employees of the ESCo do not maintain access if their accounts have not been manually disabled.

Access Control

Access control to the platform is achieved through the use of user rights. Two database tables are defined: one to contain a set of user rights and another defines a many-to-many relation by linking user rights to users. An illustration of the schema is shown in Figure 34. In order to grant a user a certain right, such as access to an application that has been implemented on the framework, an entry is created in the linking table containing the IDs for both the user and user right.

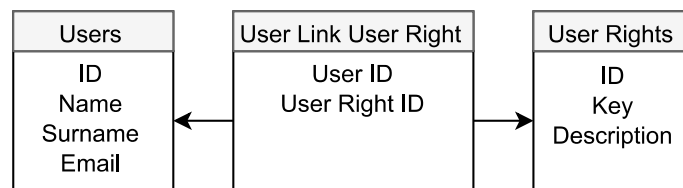


Figure 34. User right database schema

When the index page of the *Dashboard* Controller is rendered, the logged-in user's rights are queried. For each tool or application that is available on the platform, it is checked whether the user has access to that tool. If so, a link to that tool is rendered on the web page. If the user does not have access to a tool, a greyed-out version of the tool's icon is rendered without a link.

In order to prevent a logged-in user from simply accessing any tool directly via entering its URL without being linked to the required user right, an area access filter is used. This filter is attached to every action that returns a View related to a specific tool and checks if the logged-in user has sufficient access rights to the tool. If the user does not have access to the tool, an error page is returned (as discussed in section 3.5.3).

3.5.2. Route mapping

Route mapping is the concept of associating a specific URL to a Controller's action method. In ASP.NET, this is done through area registrations. In an area registration handler, URL patterns are specified. When

a request is received by the web application, all available area registrations are analysed and the first matching URL pattern is then used. The pattern specifies the following information:

- Area
- Controller
- Action
- Optional parameters
- Namespace

This allows a short and simple URL to map to an action in a long and complicated namespace. For instance, if a user requests a login page located in the “*WebApp.Areas.Users.Library.Authentication*” namespace, the default URL route will be *www.mysite.com/Users/Library/Authentication*. Instead, it may rather be mapped to a URL such as *www.mysite.com/UserLogin*.

3.5.3. Error handling

Error handling is an important part of software development. Ignoring or handling errors incorrectly may cause an application to become unstable. If an unhandled exception is encountered in ASP.NET, a generic error page is presented to the user, which provides some details regarding the error. Since this is not desired in a real-world application, custom error handling functionality has been implemented in the framework to hide these details from the user. Figure 35 shows the custom error handling process.

When an unhandled exception is detected in ASP.NET, it is passed to a built-in global function, *Application_Error*. This function is then used to analyse the exception and determine what type of error has occurred. Details regarding the error is also logged to a database, which may then be reviewed by a developer at a later stage in order to fix the issue.

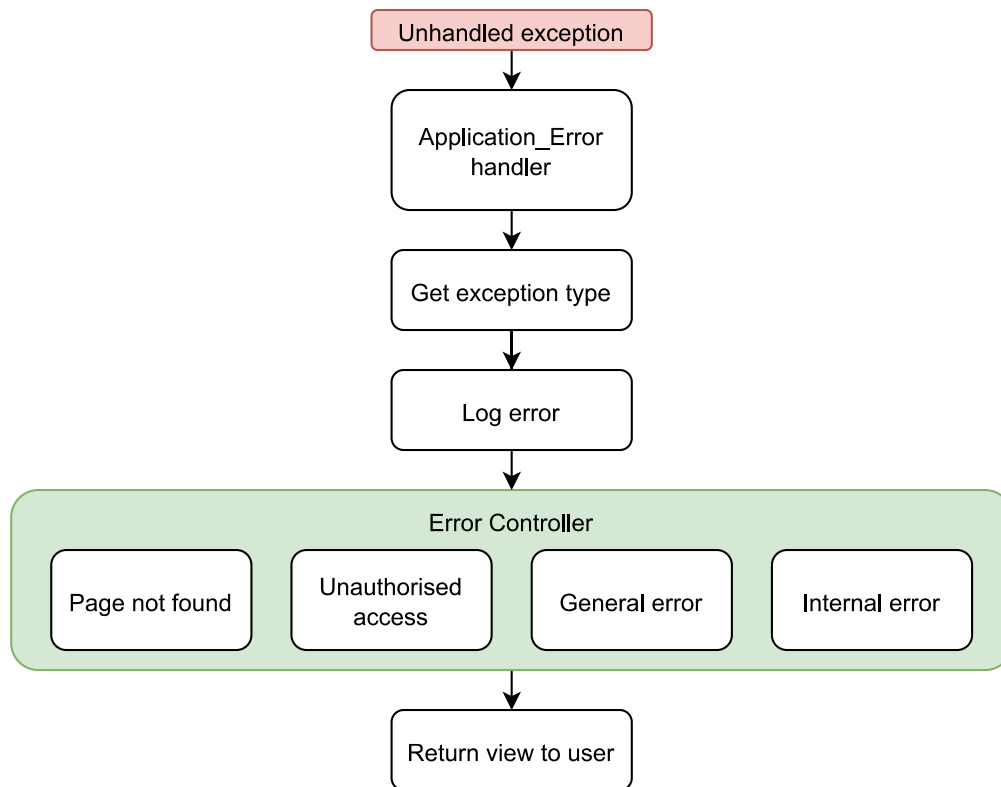


Figure 35. Custom error handling

A special error Controller is used to notify the user of the unhandled exception. Once *Application_Error* has determined which type of error has occurred, it will call a corresponding action in the error Controller. The four types of errors are:

- **Page not found** - Occurs when an unknown page is requested (HTTP error code 404)
- **Unauthorised access** - Occurs when access to a page is restricted (HTTP error code 401 and 403)
- **General error** - Occurs when an unknown HTTP error is detected
- **Internal error** - Occurs when an error is detected in the application code

Each action then returns a special View to the user, informing them that an error has occurred.

3.5.4. Components

As part of developing a modular framework, a set of components were developed to provide a selection of visual components and library elements. These components may either be used within logic processing in a Controller, or as visual elements in a View. The following list of components were developed:

- Grid
- Linking Table
- Tree
- Tabs
- Controls
- JS Utilities
- Filters
- Library
- Graphics

For more details on each component, see Appendix 1.

3.6. Development Practices

In order to keep technical debt to a minimum, certain standards and processes were followed to ensure that a high level of quality is maintained throughout the framework.

3.6.1. *Coding Standard*

In order to ensure that all code is developed in a consistent style and according to an acceptable standard, a coding standard is used. An existing coding standard defined by the ESCo is used to develop the framework for this study. The coding standard defines the following:

- A code version control tool should be used.
- How testing and code reviews should be conducted.
- Standard code libraries that may be used.
- General rules on how to format and write code.

Following the coding standard ensures that a certain level of quality is maintained. It also makes maintenance on existing code easier, since a new developer should be comfortable with the format in which the code is written.

3.6.2. Adding Web Applications on the Framework

This subsection serves as the best-practice guidelines for adding a new web application, hereafter referred to as the “app”, on the framework. The objective of these guidelines is to aid in rapid application development by providing the developer with an easy-to-follow set of steps. Figure 36 illustrates the process that should be followed to add a new app to the platform.

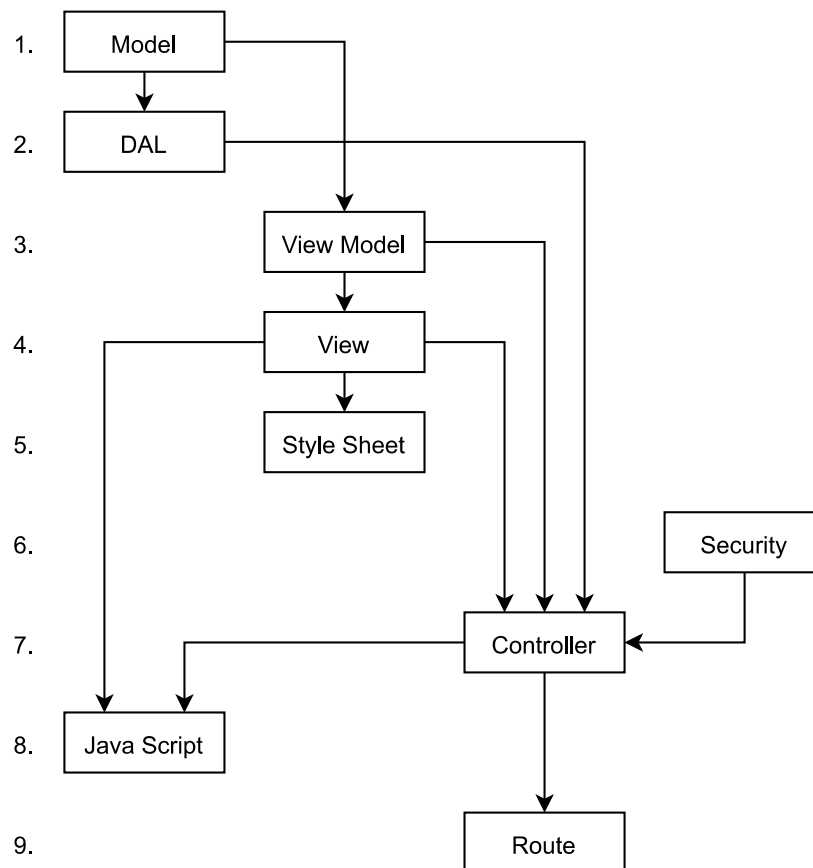


Figure 36. Guidelines for adding a web application

Model

The first step is to create Models for all database tables required for the application. Each of these Models should be placed in either the *Database* or *Custom*¹⁹ namespace, and contain properties matching the database table or SQL result.

DAL

After Models were created, a DAL should be added for each database table’s corresponding Model. Similar to the Models, a DAL should be placed in the correct namespace²⁰.

¹⁹ See section 3.4.4

²⁰ See section 3.4.5

View Model

Before a View can be created, a VM is required to encapsulate all the necessary Models and additional data required on the GUI²¹. If a web page does not require a VM, this step should be skipped. All VMs should be placed in the *ViewModels* namespace of the app's Area.

View

A View may be created once the necessary VMs are available for data binding. Once bound, the framework's layout²² View should be referenced. All sections within the layout should then be defined, such as the page title, required scripts and sidebar. At this stage the necessary JS and CSS files do not yet exist, but they should be referenced beforehand nonetheless.

After the basic page structure has been defined, additional HTML for the page should be defined. This includes all the GUI elements that the user should see and interact with. Data from the bound VM should also be rendered where necessary. A View should be placed in a folder corresponding to the Controller name that will call that View under the app's *Views* namespace.

Style Sheet

If a web page requires any custom GUI styling, such as positioning elements, or applying colour themes, a CSS file should be created for that page²³. The CSS file should be placed in the app's *Content* namespace, and correspond with the app's name.

Security

In order to maintain a secure environment, a user right should be created for the new app²⁴. This user right should then be registered with the access control filter, in order for Controller actions to use it.

Controller

All Controllers associated with the app should be created in the *Controllers* namespace. Any action that will be required from the client side should be defined, such as actions returning a web page, or query actions that should return a dataset based on a set of arguments. All required filters²⁵ (including access control filters) should also be attached to actions.

²¹ See section 3.4.2: *View Model*

²² See section 3.4.7: *View Layout*

²³ See section 3.4.7: *Style Sheets*

²⁴ See section 3.5.1: *Access Control*

²⁵ See Appendix 1: *Filters*

JavaScript

Since the web page has already been defined, a JS file is required for that page in order to add client-side functionality. All JS files should be placed within the app's *Scripts* namespace and should define any functionality required for user interaction. This includes configuring any jQuery plugins, or using any of the available framework utilities²⁶.

Route

The final step in adding a new app is to configure all the required URL routes²⁷. When the area for the app was created, the ASP.NET framework automatically created a file to define routes. This file should be updated to include routes for the app's actions according to the following format:

```
www.mysite.com/AreaName/ControllerName/ActionName
```

3.7. Framework Verification

This subsection verifies that the developed framework met the specifications set in section 3.2.

3.7.1. Database

The *MySQL.Data* NuGet²⁸ package was installed for the framework. This package adds classes and functionality required to connect to a MySQL database. The base DAL class²⁹ references these MySQL classes in order to connect, query and disconnect to a specified database schema.

3.7.2. User Authentication

Using the DAL created for the *Users* database table and the *Authenticate* action on the framework's *Login* Controller³⁰, the process in Figure 33 on page 54 was implemented. Figure 37 shows the login screen from where a user is required to enter a username and password. If the user is registered in the database and provides a correct password, they will be granted access (Figure 38). However, if the user account does not exist, or an incorrect password was entered, access is denied (Figure 39).

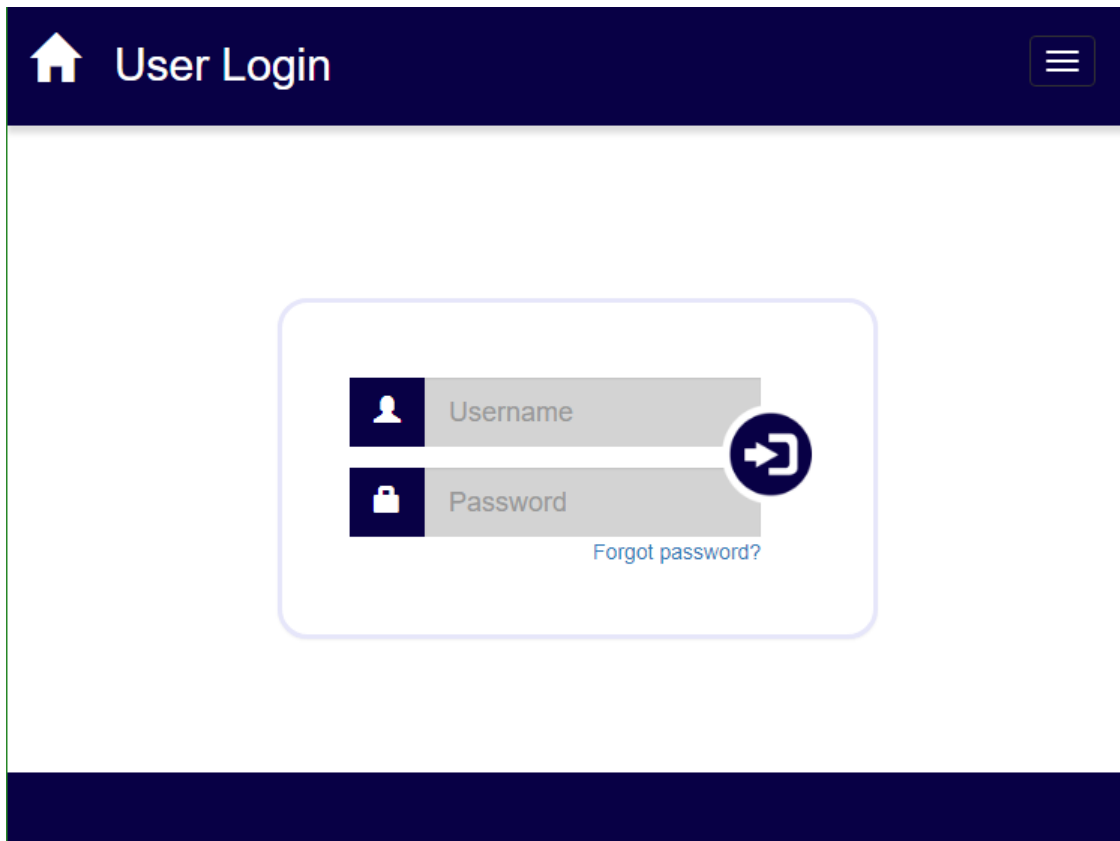
²⁶ See section 3.5.4

²⁷ See section 3.5.2

²⁸ See section 3.3.2

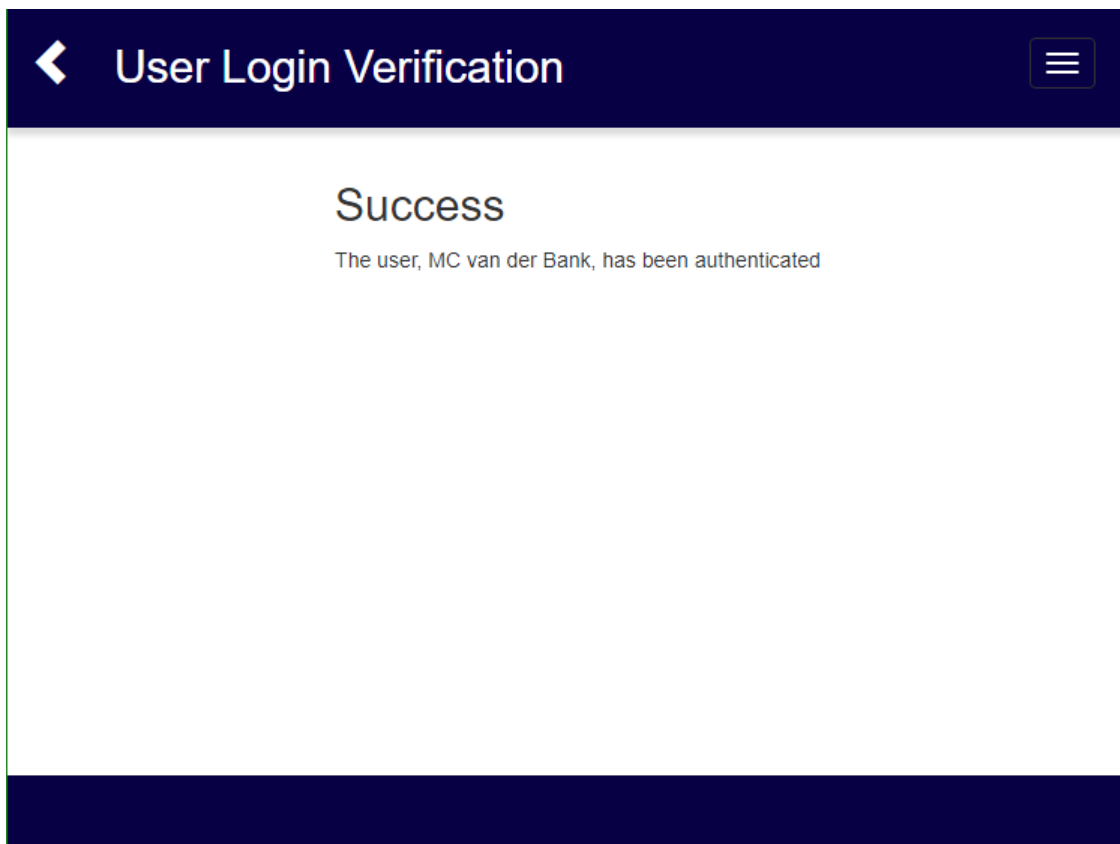
²⁹ See section 3.4.2: *Data Access Layer*

³⁰ See section 3.5.1: *Login*



The image shows a user login page with a dark blue header. On the left of the header is a white house icon followed by the text "User Login". On the right is a white hamburger menu icon. The main content area is white and contains a light purple rounded rectangle. Inside this rectangle are two input fields: the top one has a white person icon and the label "Username"; the bottom one has a white lock icon and the label "Password". To the right of these fields is a dark blue circular button with a white right-pointing arrow. Below the password field is a blue link that says "Forgot password?". The page has a dark blue footer.

Figure 37. User login page



The image shows a user login verification page with a dark blue header. On the left of the header is a white left-pointing arrow followed by the text "User Login Verification". On the right is a white hamburger menu icon. The main content area is white and contains the word "Success" in a large, bold, dark blue font. Below it, in a smaller dark blue font, is the text "The user, MC van der Bank, has been authenticated". The page has a dark blue footer.

Figure 38. Successful user login attempt

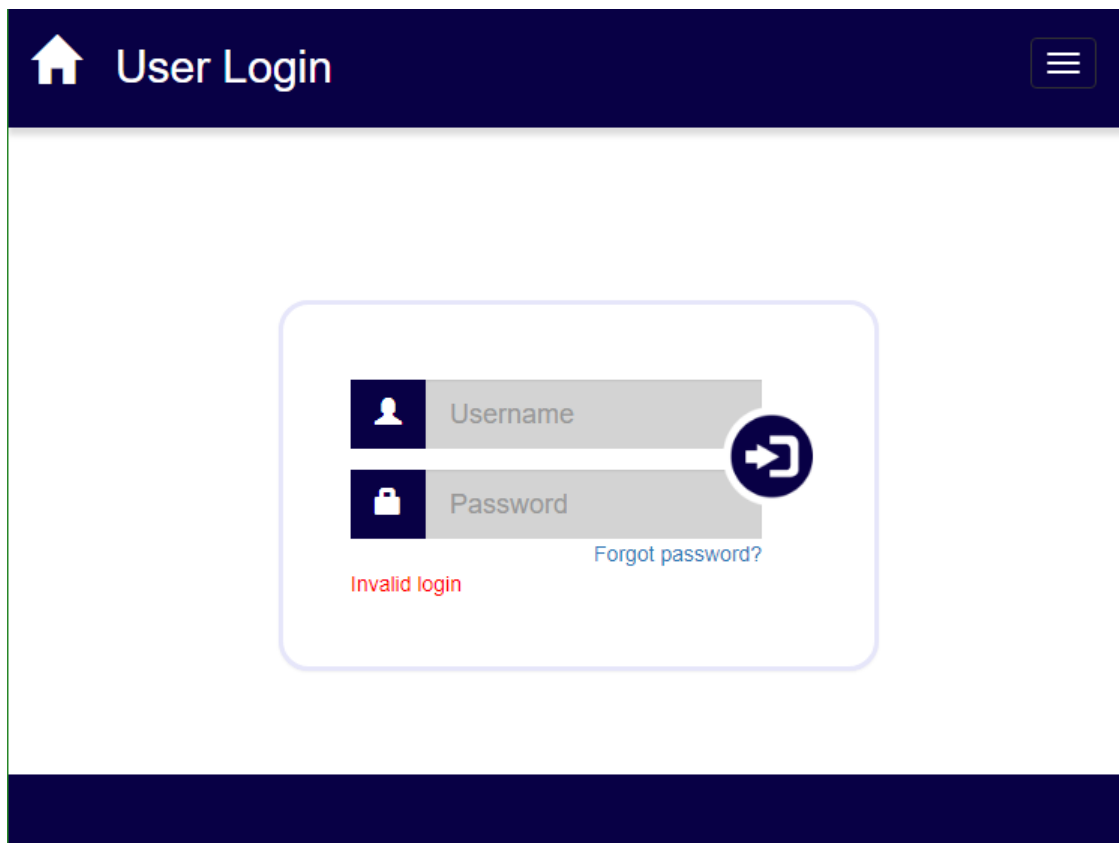


Figure 39. Failed user login attempt

In order to verify that the security feature, which automatically disables inactive user accounts, work correctly, the following process was followed:

1. Log in with a user account, which ensures that the date of the last successful login is recorded. (Figure 40)
2. Log out of the platform.
3. Open the user account table in the database, and manually subtract more than ninety days from the last successful login date (Figure 41).
4. Manually trigger a service that scans through all user accounts and disable inactive ones.
5. Attempt to log in again with the same user account.

	fk_u_ID	upw_Password	upw_Passkey	upw_Enabled	up	up	upw_LastSuccessfulLoginDate
	188	080F03E89B...	52v8kcmbio...	1	NULL	NULL	2017-07-06 14:31:07
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 40. Successfully updated last login date

	fk_u_ID	upw_Password	upw_Passkey	upw_Enabled	up	up	upw_LastSuccessfulLoginDate
	188	080F03E89B...	52v8kcmbio...	1	NULL	NULL	2017-04-01 14:31:07
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 41. Manually rewind last successful login date

The user account was unable to gain access, as shown in Figure 39.

3.7.3. Accessibility

Since a View is returned to a user's web browser in the form of an HTML document, any modern web browser on any operating system can access the platform. By using Bootstrap³¹, mobile web browsers on smartphones and tablets are also able to view a scaled-down version of the platform, as shown in Figure 42.

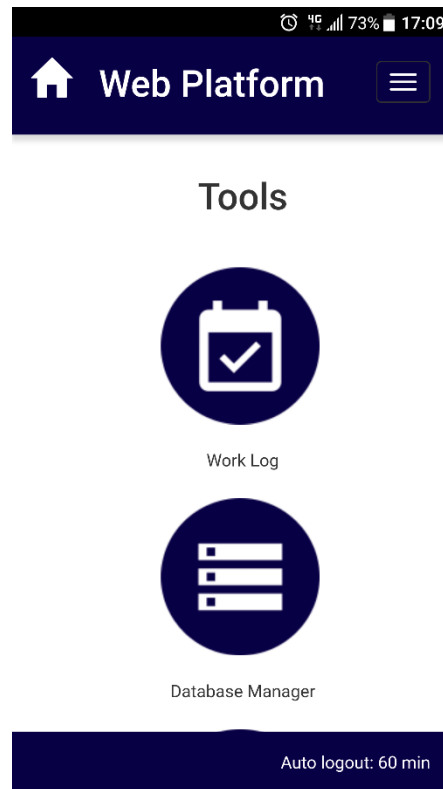


Figure 42. Mobile browser layout

However, due to the nature of some applications, it is not always possible to render them perfect for a device with a smaller screen. Figure 43 shows the same web page rendered on a desktop web browser.

³¹ See section 3.3.2: *Bootstrap*

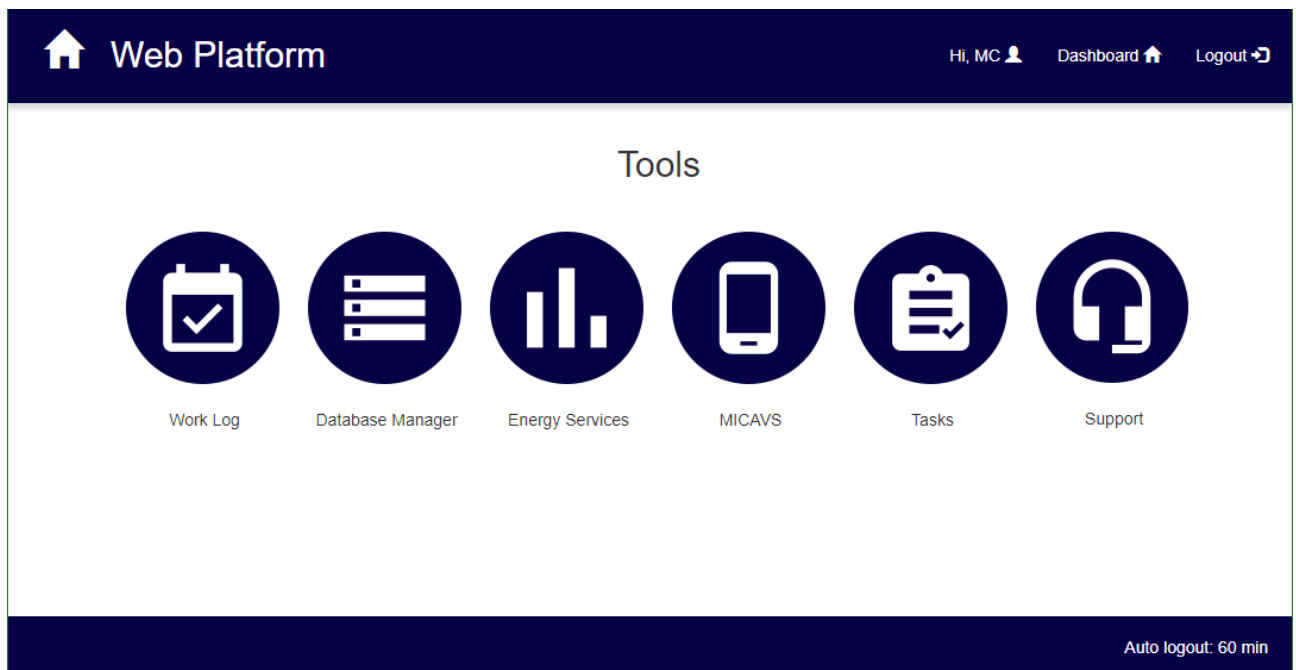


Figure 43. Desktop browser layout

3.7.4. Access Control

System security is achieved through an authentication process and access control filters³². The following figures provide a visual representation of how user rights affect access control. In order for a user account to log into the platform and gain access to the dashboard, a user right called “Platform access” should be granted, as illustrated in Figure 44.

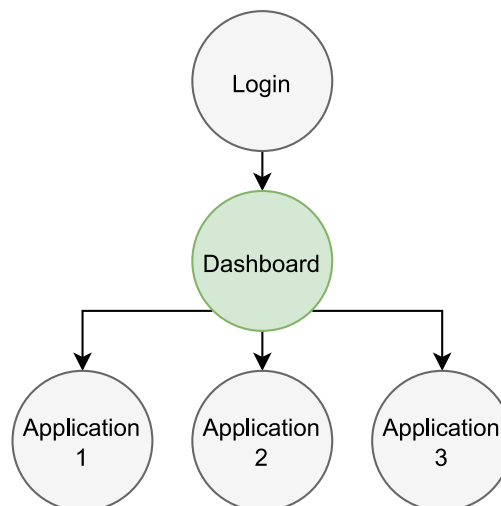


Figure 44. User right that grants access to the dashboard

Additionally, in Figure 45, before the user may use an application on the platform, they should also be granted a user right specifically for that application.

³² See section 3.5.1

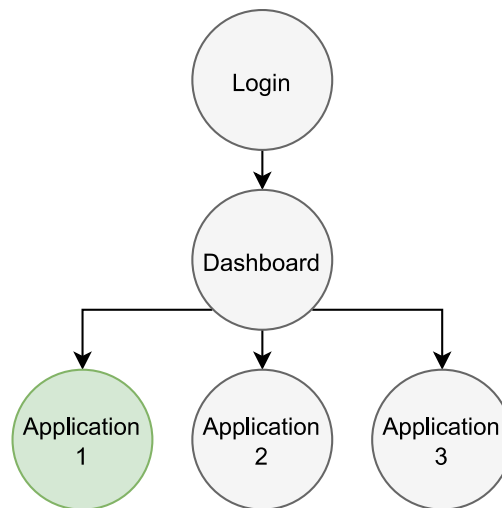


Figure 45. User right that grants access to application 1

In order to verify that the authentication and access control process is working as expected, attempts were made to access web pages on the platform via their URLs without logging in, or without sufficient permissions. Figure 46 shows an example where a specific URL for an application was requested by a user account without logging in beforehand.

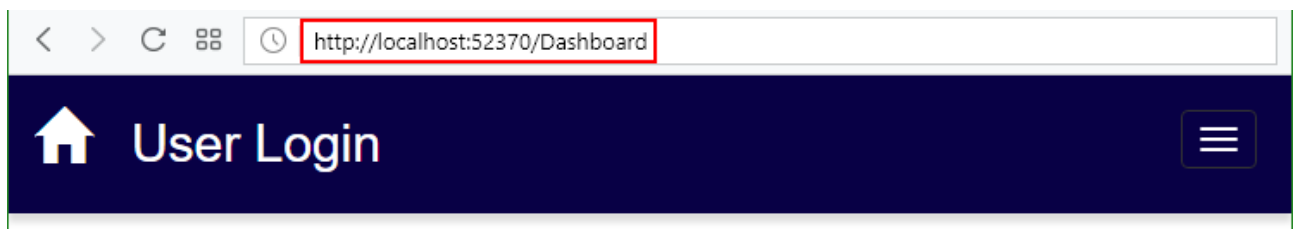


Figure 46. Access URL without logging in

Similarly, Figure 47 shows an example where an already logged-in user attempted to access a URL of an application to which they do not have access to.



Figure 47. Access URL with insufficient rights

In both cases above, the user was redirected to the framework's custom error page³³, as shown in Figure 48.

³³ See section 3.5.3

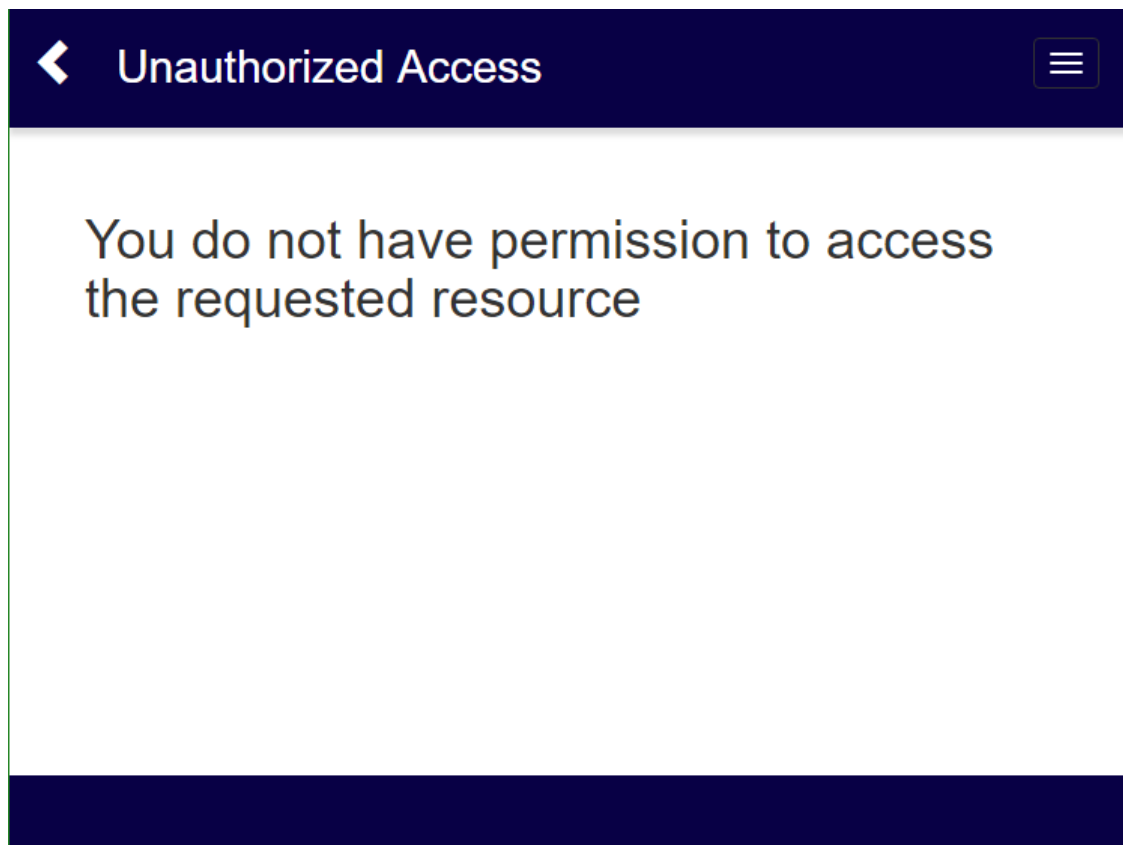


Figure 48. Unauthorised access error

3.8. Design Summary

This chapter discussed the design and implementation of a modular framework to achieve effective development of web applications. The framework is based on the MVC architecture, which was identified from the literature in chapter 2, and includes adaptations to improve efficiency and modularity. A best-practice guideline was also developed, which outlines the process that should be followed when using the framework for new development.

The developed framework was also verified in terms of the specified requirements to ensure that all specifications were met. This included the authentication of user accounts in an existing MySQL database, cross-platform accessibility, and access control on user accounts to prevent unauthorised access to specified resources.

Chapter 4 will validate the developed framework to ensure that the problem statement and objectives were achieved. A case study is performed to validate the modularity of the framework, and a survey conducted to ensure that various other objectives are also met.

Chapter 4

Results and Case Studies

4.1. Preamble

This chapter will validate the platform developed in this study in terms of the problem statement presented in section 1.4. The platform was designed and implemented in Chapter 3 and provides a framework which can be used to rapidly develop a set of web applications.

Modularity will be proven by examining examples where existing functionality has been altered. In order to validate that the framework accelerates development, simplifies maintenance and reduces technical debt, a case study was performed by implementing a series of web applications. The case study will evaluate the effectiveness of the framework, and assess whether or not the problem statement was solved. The assessment will be done by a survey, where software developers were asked to answer questions regarding the development process.

A reduction in technical debt is validated by performing an analysis on the platform with the use of third-party analysis software. An additional analysis was performed on an existing web platform in order to provide a baseline for the results.

For the purposes of the survey and technical debt analysis in section 4.4 to 4.6, the framework/platform developed in this study will be referred to as “Platform A”. An existing web system of the ESCo will be referred to as “Platform B”, and is used as a baseline, or point of reference, when analysing results.

4.2. Modularity

This subsection aims to validate the framework in terms of the following goal stated in section 1.5:

*Must be **modular** by providing simple and reusable components, which can be used to **rapidly develop** new functionality.*

From the detailed design in section 3.4, it can be concluded that the framework is indeed modular. By utilising a View layout³⁴ and reusable components³⁵, new web pages can be rapidly developed and added to new or existing web applications on the platform.

In order to assess whether or not changes can be made with ease to existing implementations, the following scenario is presented. An existing user interface contains a grid (Figure 49) that is populated with data from a table in the database, as shown in Figure 50.

³⁴ See section 3.4.7

³⁵ Including but not limited to components discussed in section 3.5.4 and Appendix 1

Example Users

Hi, MC Dashboard Logout

Auto logout: 60 min

	Name	Surname	Age	Email Address	
≡	Blair	Towne	29	Blair.Towne@example.com	X
≡	Carlota	Grisby	25	Carlota.Grisby@example.com	X
≡	Darren	Lanahan	65	Darren.Lanahan@example.com	X
≡	Eli	Ugarte	54	Eli.Ugarte@example.com	X
≡	Ima	Kircher	34	Ima.Kircher@example.com	X
≡	Jaleesa	Heaps	49	Jaleesa.Heaps@example.com	X
≡	Joanna	Herrick	38	Joanna.Herrick@example.com	X
≡	Jospeh	Yamamoto	51	Jospeh.Yamamoto@example.com	X
≡	Myrta	Oller	43	Myrta.Oller@example.com	X
≡	Wilma	Overstreet	35	Wilma.Overstreet@example.com	X

Results per page 20 1

Figure 49. Existing user interface example

	ID	Name	Surname	Age	EmailAddress
	1	Wilma	Overstreet	35	Wilma.Overstreet@example.com
	2	Joanna	Herrick	38	Joanna.Herrick@example.com
	3	Blair	Towne	29	Blair.Towne@example.com
	4	Mvrta	Oller	43	Mvrta.Oller@example.com
	5	Eli	Uoarte	54	Eli.Uoarte@example.com
	6	Darren	Lanahan	65	Darren.Lanahan@example.com
	7	Ima	Kircher	34	Ima.Kircher@example.com
	8	Carlota	Grisbv	25	Carlota.Grisbv@example.com
	9	Jospeh	Yamamoto	51	Jospeh.Yamamoto@example.com
	10	Jaleesa	Heaps	49	Jaleesa.Heaps@example.com

Figure 50. Example data

It is then required to add an additional column to the table and display this new column on the user interface as well. The following process was performed in order to achieve this:

1. Add the new column in the database table
2. Add a new property to the model representing the table
3. Update the DAL to include the new property
4. Add a new column in the grid's Partial View

Figure 51 shows the new column that was added to the database table.

	ID	Name	Surname	Age	EmailAddress	Phone
	1	Wilma	Overstreet	35	Wilma.Overstreet@example.com	156489237
	2	Joanna	Herrick	38	Joanna.Herrick@example.com	564894135
	3	Blair	Towne	29	Blair.Towne@example.com	454684512
	4	Mvrta	Oller	43	Mvrta.Oller@example.com	287454217
	5	Eli	Uoarte	54	Eli.Uoarte@example.com	561584545
	6	Darren	Lanahan	65	Darren.Lanahan@example.com	478788212
	7	Ima	Kircher	34	Ima.Kircher@example.com	132848783
	8	Carlota	Grisbv	25	Carlota.Grisbv@example.com	215684865
	9	Jospeh	Yamamoto	51	Jospeh.Yamamoto@example.com	184864318
	10	Jaleesa	Heaps	49	Jaleesa.Heaps@example.com	215445842

Figure 51. Add the new column in MySQL

The Model is expanded to include the new phone number property in Figure 52.

```
public class ExampleUserM
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
    public int Age { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
}
```

Figure 52. Add a new property to the Model

Figure 53 illustrates the new binding that is configured in the DAL. The binding statement shown ensures that the values from the database table is able to load into the Model's "Phone" property.

```
public override object MapTo(MySqlDataReader aData)
{
    ExampleUserM lEntry = new ExampleUserM();

    Set(aData, ref lEntry, "ID", "ID");
    Set(aData, ref lEntry, "Name", "Name");
    Set(aData, ref lEntry, "Surname", "Surname");
    Set(aData, ref lEntry, "Age", "Age");
    Set(aData, ref lEntry, "Email", "Email");
    Set(aData, ref lEntry, "Phone", "Phone");

    return lEntry;
}
```

Figure 53. Add SQL mapping to DAL

The SQL query is also updated in the DAL to ensure that that query will correctly handle the new Phone column, as shown in Figure 54.

```

public override bool CreateEntry(ExampleUserM aNewItem)
{
    string lQuery = "INSERT INTO Example_Users (Name, Surname, Age, EmailAddress, Phone) " +
        "VALUES (@Name, @Surname, @Age, @EmailAddress, @Phone)";

    var lParameters = new List<KeyValuePair<string, object>>();
    lParameters.Add(new KeyValuePair<string, object>("@Name", aNewItem.Name));
    lParameters.Add(new KeyValuePair<string, object>("@Surname", aNewItem.Surname));
    lParameters.Add(new KeyValuePair<string, object>("@Age", aNewItem.Age));
    lParameters.Add(new KeyValuePair<string, object>("@EmailAddress", aNewItem.Email));
    lParameters.Add(new KeyValuePair<string, object>("@Phone", aNewItem.Phone));

    try
    {
        ExecuteQuery(lQuery, lParameters);
    }
    catch (Exception e)
    {
        throw new Exception(e.Message);
    }

    return true;
}

```

Figure 54. Add SQL statement to DAL

Finally, in Figure 55 the View is updated to include a new column in the HTML table, and to load data into all rows.

```

<table>
  <tr>
    .
    .
    <th>
      Email Address
    </th>
    <th>
      Phone
    </th>
  </tr>

  @if (Model.Entries != null && Model.Entries.Count > 0)
  {
    for (int i = 0; i < Model.Entries.Count; i++)
    {
      <tr>
        .
        .
        <td>
          @Html.TextBoxFor(o => o.Entries[i].Email)
        </td>
        <td>
          @Html.TextBoxFor(o => o.Entries[i].Phone)
        </td>
      </tr>
    }
  }
</table>

```

Figure 55. Add column on View

Figure 56 shows that the user interface included the new column after these changes were made.

	Name	Surname	Age	Email Address	Phone	
≡	Blair	Towne	29	Blair.Towne@example.com	454684512	X
≡	Carlota	Grisby	25	Carlota.Grisby@example.com	215684865	X
≡	Darren	Lanahan	65	Darren.Lanahan@example.com	478788212	X
≡	Eli	Ugarte	54	Eli.Ugarte@example.com	561584545	X
≡	Ima	Kircher	34	Ima.Kircher@example.com	132848783	X
≡	Jaleesa	Heaps	49	Jaleesa.Heaps@example.com	215445842	X
≡	Joanna	Herrick	38	Joanna.Herrick@example.com	564894135	X
≡	Jospeh	Yamamoto	51	Jospeh.Yamamoto@example.com	184864318	X
≡	Myrta	Oller	43	Myrta.Oller@example.com	287454217	X
≡	Wilma	Overstreet	35	Wilma.Overstreet@example.com	156489237	X

Figure 56. Updated user interface with new column

Modularity can be validated by examining an example that would require a change to a certain aspect of the framework, without affecting other components that depend on it. Such an example would be to replace the current Relational Database Management System (RDBMS) (such as MySQL) with either another RDBMS, such as Microsoft SQL Server, or completely different database technology. Figure 57 shows the standard architecture, where the base DAL connects to the MySQL database and provides functionality to read and write data.

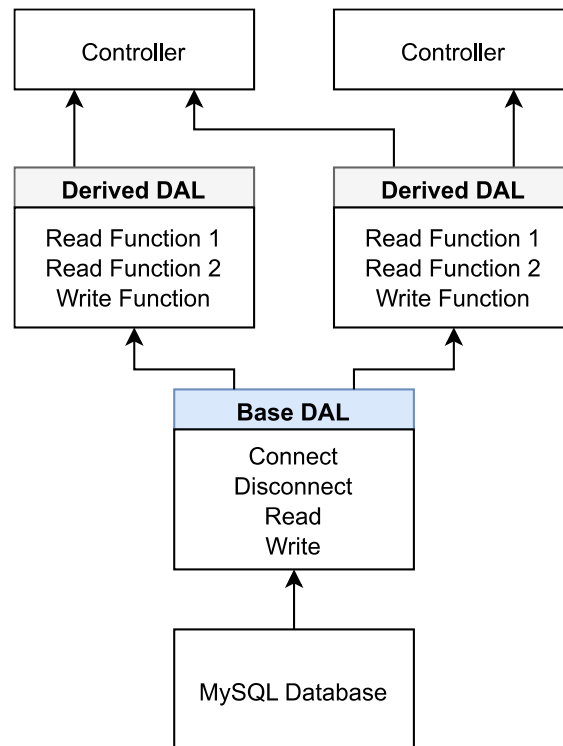


Figure 57. Standard architecture

If the MySQL database should be replaced with Microsoft SQL Server, the only architectural changes required will be to the base DAL (see Figure 30 on page 50), since the queries in the derived DAL classes should stay the same, as shown in Figure 58. Therefore, all other areas of the framework that depend on the derived DAL classes (such as the Controllers) will remain unaware of the changes that took place.

However, if an entirely different database technology is to be used, the derived DAL classes will also be affected. For example, in Figure 59 a document store database such as MongoDB³⁶ does not use the same query syntax as an RDBMS, and therefore the base DAL and derived DAL classes should be replaced. Care should be taken when reimplementing the derived classes to ensure that they still contain the same number of functions with the same function definitions for the rest of the framework to remain unaffected.

³⁶ MongoDB is a big data solution. For more information, see <https://www.mongodb.com/what-is-mongodb>

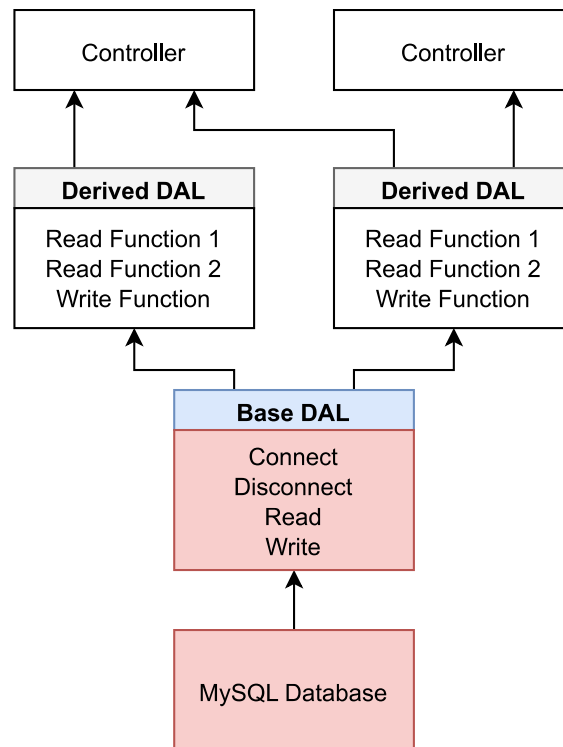


Figure 58. Modified architecture for Microsoft SQL Server

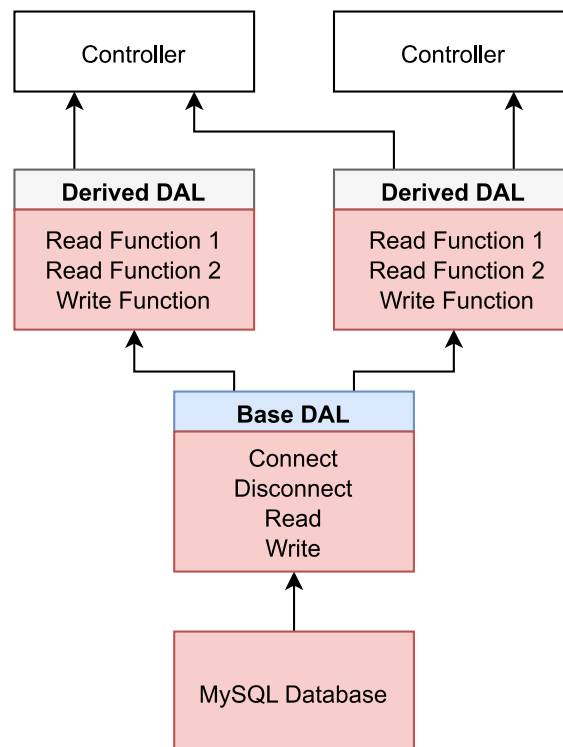


Figure 59. Modified architecture for MongoDB

Summary

The validation provided in this subsection argues that the objective in terms of modularity (as stated in section 1.5) was achieved. Rapid development was only partially proven due to the limited scope of the example presented here. Further validation in terms of this is presented in section 4.5.

4.3. Case Study

A case study was conducted by implementing three web applications with the framework. Each application that was developed with the framework used the best-practice guideline process illustrated in section 3.6.2. The experience gained from implementing these applications will allow observations to be made in order to evaluate the effectiveness of the framework (in section 4.4 and 4.5). Each application will be briefly discussed in terms of its purpose and guidelines that were followed.

In addition to providing a base from which to observe the effectiveness of the framework, the reusability of framework components is also validated, as required in section 1.5:

*Must be modular by providing simple and **reusable components** which can be used to rapidly develop new functionality.*

Each application developed with the framework will analyse how many lines of code were reused from existing features and components versus new functionality developed.

4.3.1. Application A – Work Logger

The purpose of this application is to provide a tool which can be used by employees of a company to keep track of all work-related activities. Any activity, such as a task completed or meeting held, is recorded on the tool. Details with regards to the activity, such as date, hours spent, or activity category, are specified and saved in a database. The following features were required of the tool:

- An employee's supervisor should be able to view their logs and sign off on all activities.
- Management employees should be able to see an overview of all the other employees' logged activities.
- A simple time summary should be displayed for each employee, showing their total time logged for the day, as well as the week.
- An activity should be associated with both a category and description.

Table 4 below shows which steps from the process in section 3.6.2 were followed.

Step	Description	Implemented?
1	Model – Models were implemented	Yes
2	DAL – DALs were implemented	Yes
3	VM – VMs were implemented	Yes
4	View – Views were implemented	Yes
5	CSS – Style sheets were implemented	Yes
6	Security – User rights were implemented	Yes
7	Controller – Controllers were implemented	Yes
8	JS – Java scripts were implemented	Yes
9	Route – Routes were added	Yes

Table 4. Guidelines followed for Work Logger

Figure 60 graphically illustrates which of the components in the framework were used. Components used are highlighted in yellow in the figure. In this case, most of the available components were used. The application did not require any functionality found in the framework libraries, nor any VMs other than its own.

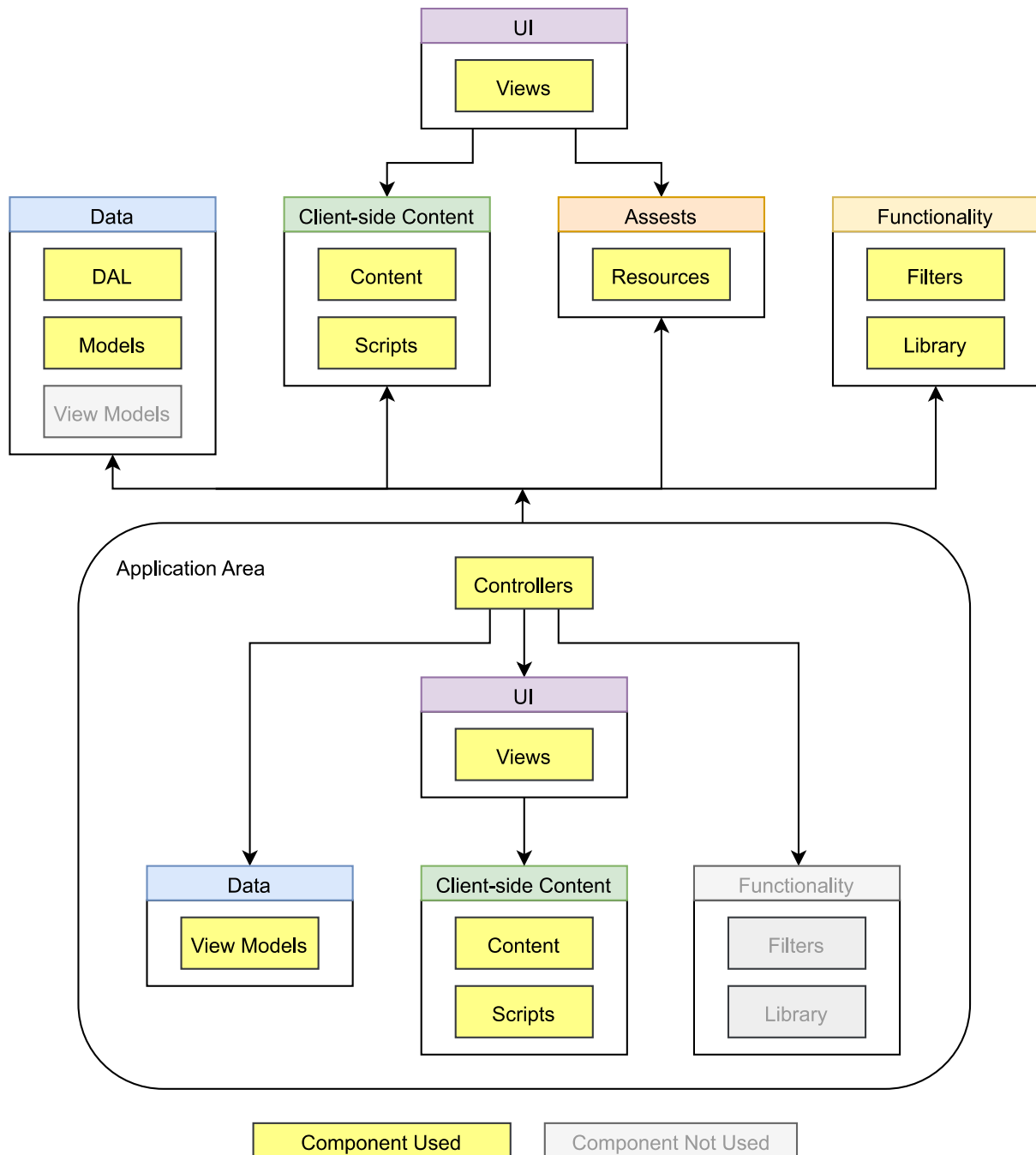


Figure 60. Application A framework component usages

Figure 61 illustrates the amount of code that was reused from existing parts of the framework, as well as code that was newly developed for the application. Table 5 provides further details by showing exactly how many lines of code were used and written in each section of the platform.

Component	Lines of Code	
	Framework Code	Application Code
Content	711	25
Controllers	0	352
DAL	2636	0
Filters	90	0
Library	656	0
Models	181	0
Scripts	3454	1040
View Models	0	68
Views	103	405
Total lines of code	7831	1890

Table 5. Application A code usage

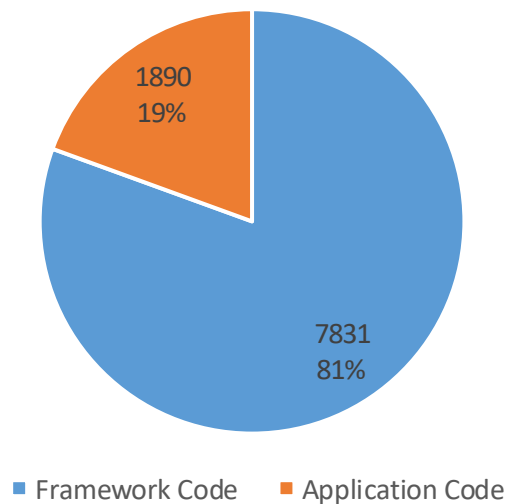


Figure 61. Lines of code distribution in Application A

In the case of this application, 81% of the overall code was reused from existing parts of the framework. The biggest contribution of framework code was from the DAL and scripts. This clearly indicates that a significant amount of time and effort was saved by reusing existing code.

4.3.2. Application B – Database Manager

This application is used as a user-friendly GUI to manage a collection of database tables. Each table in the database schema is represented by a single “menu” in the tool. The user is presented with an initial dashboard containing a selection of menus. Each menu then provides basic CRUD operations for that table.

Two forms of menus exist: a grid menu and a linking menu. A grid menu represents a table “as is”, where all the columns are displayed and the user may add, remove or edit content. A linking menu

provides an interface to manage a many-to-many relation database table, as illustrated in Figure 34 on page 55. For the example shown in this figure, a *user* is selected from a selection box. Two columns are then populated with linked and unlinked *user rights*. The user then moves items between columns to either link or unlink them from the *user*.

Special menus that combine the functionality of multiple tables were also implemented. An example of such is a menu to manage user accounts. Multiple tables related to user accounts, such as passwords, user rights, or additional information, are presented on a single page. All operations performed in this menu apply to a selected user account.

All the steps in section 3.6.2 were followed in Table 6 below.

Step	Description	Implemented?
1	Model – Models were implemented	Yes
2	DAL – DALs were implemented	Yes
3	VM – VMs were implemented	Yes
4	View – Views were implemented	Yes
5	CSS – Style sheets were implemented	Yes
6	Security – User rights were implemented	Yes
7	Controller – Controllers were implemented	Yes
8	JS – Java scripts were implemented	Yes
9	Route – Routes were added	Yes

Table 6. Guidelines followed for Database Manager

As shown in Figure 62, all the framework components were used. New code that was developed for the application is also shown, and no new filters or libraries, other than those of the framework, were required.

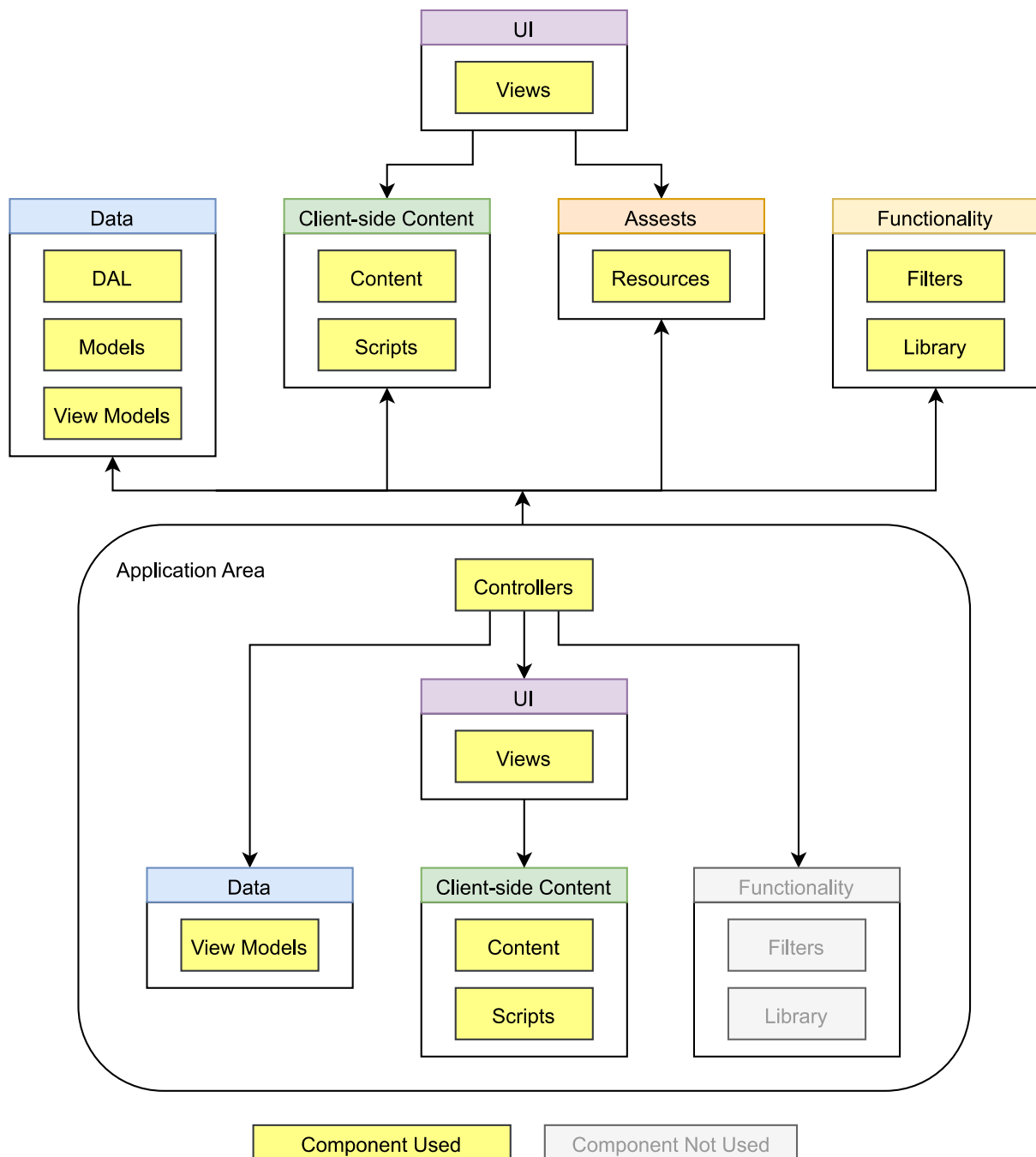


Figure 62. Application B framework component usages

Table 7 shows a detailed overview of the amount of code that was reused from existing components in the framework, as well as new code that was developed. Figure 63 summarises this information in a chart and illustrates the amount of code as a percentage.

Component	Lines of Code	
	Framework Code	Application Code
Content	864	106
Controllers	0	8075
DAL	9448	0
Filters	184	0
Library	1080	0
Models	850	0
Scripts	4927	5099
View Models	28	628
Views	103	3542
Total lines of code	17484	17450

Table 7. Application B code usage

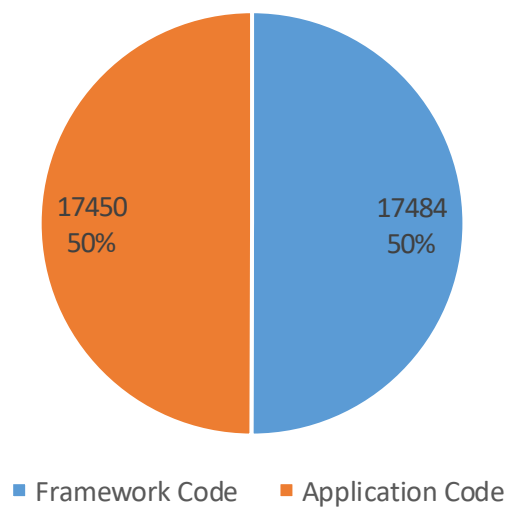


Figure 63. Lines of code distribution in Application B

An equal distribution of code usage was observed. The majority of reused framework code was from the DAL and scripts, where new code was primarily Controllers, scripts and Views. This outcome was expected, since the Database Manager application relies heavily on connections to all the database tables, in which case framework DAL code had to be used. Controllers, scripts and Views also make up the most significant parts that are used for GUIs.

4.3.3. Application C – Services

In some cases there is a need for external systems to communicate with the platform. This may include a desktop or mobile application that requires data from the database, or if a series of events should be triggered. This application provides Application Programming Interface (API) endpoints³⁷ for various

³⁷ An endpoint is a URL where a service may be accessed by a client application.

external applications that require access to the system. Since these services will only perform data transactions, there is no need for a GUI or any client-side content.

Table 8 below indicates that only data and logic-related steps were implemented, since no GUI is required for this application.

Step	Description	Implemented?
1	Model – Models were implemented	Yes
2	DAL – DALs were implemented	Yes
3	VM – VMs were implemented	No
4	View – Views were implemented	No
5	CSS – Style sheets were implemented	No
6	Security – User rights were implemented	Yes
7	Controller – Controllers were implemented	Yes
8	JS – Java scripts were implemented	No
9	Route – Routes were added	Yes

Table 8. Guidelines followed for Services

The simplified structure required for services also meant that only a few parts of the framework were required, as shown in Figure 64. No GUI-related code such as Views, Content or Scripts were developed or reused. Only data-related and other functional code were used.

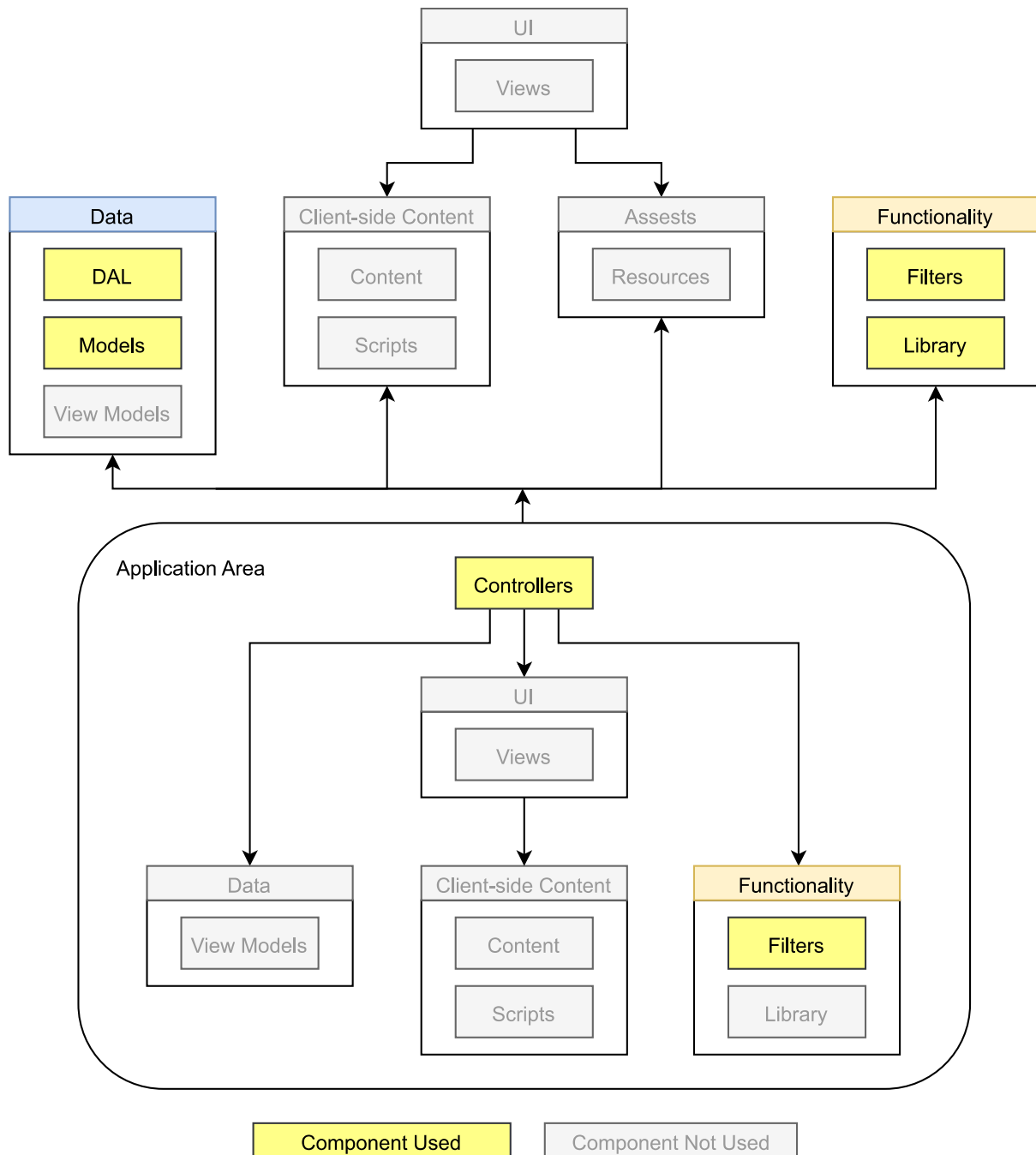


Figure 64. Application C framework component usages

A detailed breakdown of code usage is shown in Table 9 and Figure 65. A significant part of the application was new library code that was developed.

Component	Lines of Code	
	Framework Code	Application Code
Content	0	0
Controllers	0	2791
DAL	2260	0
Filters	276	0
Library	2784	5761
Models	194	0
Scripts	0	0
View Models	0	0
Views	0	0
Total lines of code	5514	8552

Table 9. Application C code usage

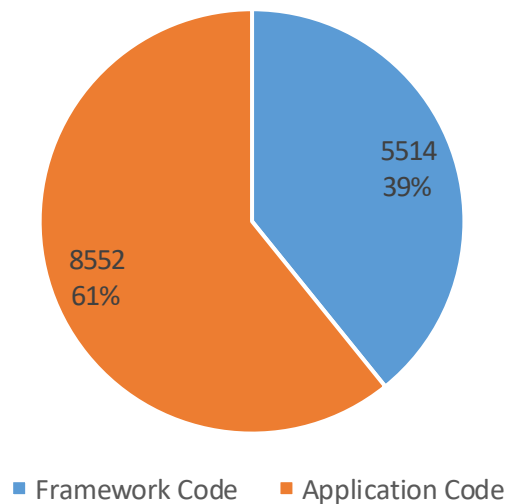


Figure 65. Lines of code distribution in Application C

Only 39% of the code used by this application was reused, with DALs and existing libraries accounting for most of it. Along with the Controllers to accept web requests for the services, new libraries to process the requests were developed.

4.3.4. Summary

Upon examination of Table 5, 7 and 9, a few trends have been observed. First, since the goal of the DAL is to contain all the platform's database-related transactions, most of the applications in this case study use a significant amount of code from the DAL when compared to code reused from other areas of the framework.

It can also be seen that applications A and B do not implement any DAL, Filter, Library or Model in its application code, but only use them from the framework. Both application A and B's most used framework code lies in the DAL and Scripts.

Application C is a special case, where it does not implement a GUI, therefore no framework Scripts are used. It relies mostly on the framework DAL and Library code, and implements its own Libraries as the biggest part of its application code. The following conclusions can be drawn:

- A significant part of any application will be reused framework DAL code.
- If an application implements a GUI, it will reuse a large amount of code in framework scripts.

The objective in terms of “reusable components” stated in section 1.5 was therefore validated.

4.4. Developer Survey

Determining how easy it is to develop for a specific system or framework depends on various factors, including, but not limited to:

- Skill and experience with the programming language
- Experience with the system or framework
- Understandability of the system or framework
- Available documentation for the system or framework

Even if a developer is skilled in the programming language and has sufficient experience with the system or framework, development can still be difficult. This may be caused by an illogical or disorganised structure, or lack of documentation to provide support. However, while these factors may provide an indication of how easy development is perceived from a human perspective, it is not readily quantifiable.

A survey was therefore performed to quantify how software developers experienced the framework developed in this study. Three of the goals set in section 1.5 are validated by the survey:

- *Must be modular by providing simple and reusable components which can be used to **rapidly develop new functionality**.*
- *Ensure that **maintenance** may be performed in an **efficient** manner.*
- *Be considered an **acceptable development framework** by software developers.*

The survey is split into two sections, each consisting of 24 identical questions across five categories. Each category evaluates a certain aspect of the framework. Questions are answered by marking a value on a numeric scale, which indicates how the person rates the question, or how much they agree with the statement. Each section is aimed towards a specific platform, where the first section evaluates Platform A, and the second Platform B. As defined on page 68, the framework/platform developed in this study will be referred to as “Platform A”, while the ESCo’s existing web system is referred to as “Platform B”. The ultimate goal of the survey is to rate or score the new framework better than the existing one.

The following categories in Table 10 below are analysed by the survey.

Category	Evaluation
Background information	Provides background information on the participant’s development experience.
Structure	Evaluates the structure of the platform in terms of architecture and system components.
Maintenance	Evaluates the platform in terms of code maintainability.
New development	Evaluates the platform in terms of new development.
Code quality	Evaluates the perceived quality of the code and technical debt in the platform.
Development environment	Evaluates the development environment and language of the platform.

Table 10. Survey categories

A summary of the questions asked in the survey follows below.

Background Information

- a) How many months of development experience do you have with this platform?
- b) How many different parts of the platform did you work on? In other words, do you have a large variety of experience, or only know how a few parts of the platform work?
- c) Do you enjoy developing for the platform?

- d) Do you feel that development on the platform provides you with a valuable learning opportunity?

Structure

1. How much time does it take to learn and understand the platform architecture?
2. How easy is it to read and follow existing code when attempting to understand a section of the platform?
3. How many reusable components exist within the platform?
4. Do existing platform components provide a wide range of functionality?
5. How much documentation is available for platform components?
6. How much documentation is available for the core programming language of the platform architecture?

Maintenance

7. When a bug is identified, how easy is it to reproduce the bug?
8. How much debugging information is available?
9. When a bug is identified, how easy is it to locate the problematic source code?
10. When fixing a user interface bug, how much other code needs to change?
11. When fixing an architectural/platform bug, how much other code needs to change?

New Development

12. How easy is it to develop new functionality using the platform architecture?
13. How long does it take to develop new functionality for the platform?
14. How easy is it to use an existing platform component?
15. How many debugging tools are available during development?
16. Is there a clear design pattern to follow when developing new functionality for the platform?

Code Quality

17. Is existing framework code in libraries and components sufficiently commented?
18. Does existing code in the platform follow a coding standard?
19. Is there a general sense of neatness and order present throughout the platform?
20. When a complex bug is identified, is more time spent to investigate a permanent fix, or less time to apply a temporary fix?
21. Does platform functionality trend towards static hard coding, or dynamic and functional?

Development Environment

22. How many third-party libraries and add-ons are available to either augment the IDE, or provide functionality for the platform?
23. How many online resources related to the platform architecture and language are available?
24. Does the IDE provide sufficient tools to support development (such as refactoring, find usages, navigation, etc.)?

The **background information** section aims to bring the experience of the developer into scope with the rest of the survey. This information may be used to explain some outlier responses to some answers. For instance, a developer with much more experience on a system may find it easier to add new functionality or fix bugs than a developer with little experience.

The **structure** section evaluates the system in terms of the overall design and analysis. It is typically desirable of a framework to feature a large amount of reusable functionality, which not only saves development time, but also improves modularity. Documentation would ideally list the available functionality of a framework, and also greatly aid a developer to understand how various framework components work and how to use them.

The **maintenance** section evaluates how effective a developer can maintain the system by fixing bugs. If a system is easy to maintain, there would ideally be enough debugging information available to aid the developer in locating the incorrect section of code. A bug is also generally easier and quicker to fix if there is a minimal level of dependency on that section of code.

The section with regards to **new development** evaluates how easy it is to add new functionality to the system. If a framework provides a wide variety of functionality with sufficient documentation (as covered by the section on the structure), it would be easier and quicker to add new functionality. A clear design pattern or process also aids with rapid development.

The **code quality** section evaluates the perceived level of quality of the system. A system with a higher level of quality is easier to maintain. Following a coding standard and sufficiently commenting code aid developers to better and quicker understand a section of code.

Finally, the **development environment** section evaluates the tools which are available to a developer, such as the IDE, third-party libraries and resources for the programming language.

4.5. Survey Results

The results from the survey forms completed by a selection of software engineers are included in Table 15 in Appendix 2. Figure 66 shows the compiled results from the survey. Only seven respondents could complete the survey due to the limited number of software developers employed by the ESCo. While this might be considered insufficient, the results from the survey will at least provide an indication of how the framework performed. Of the seven respondents, one didn't complete the section for Platform A due to inexperience with that particular platform.

Questions were answered by indicating a number on a scale. The number indicates how much a person agrees with the question or statement, and the scale for each question is defined. A higher number indicates a more desirable outcome for the answer, and all question scales are defined, where a value of 5 is the best answer. Figure 66 shows the average score obtained by each question.

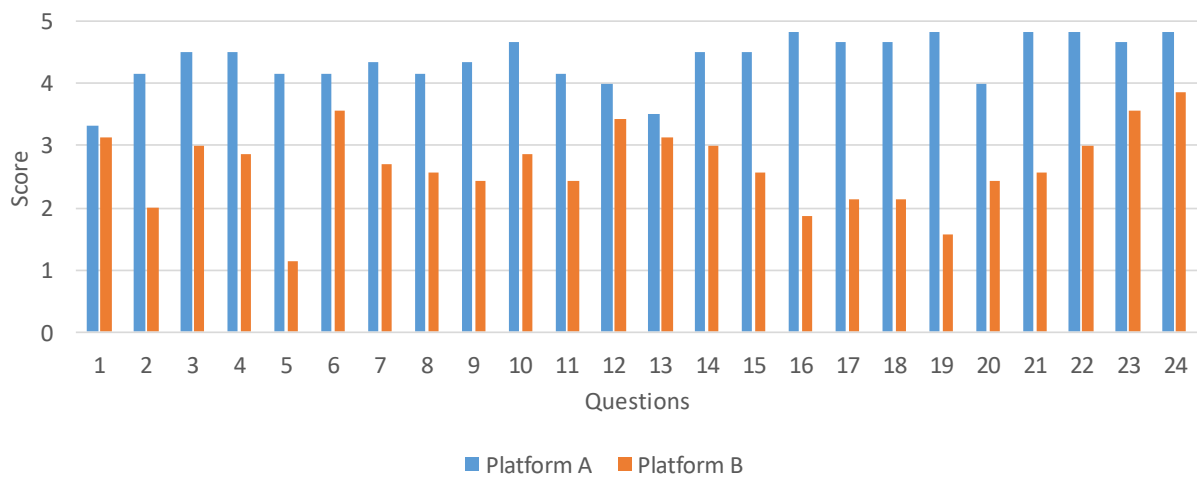


Figure 66. Developer survey results

It is evident from Figure 66 that Platform A performed better on all questions. Figure 67 compares the average scores of each section.

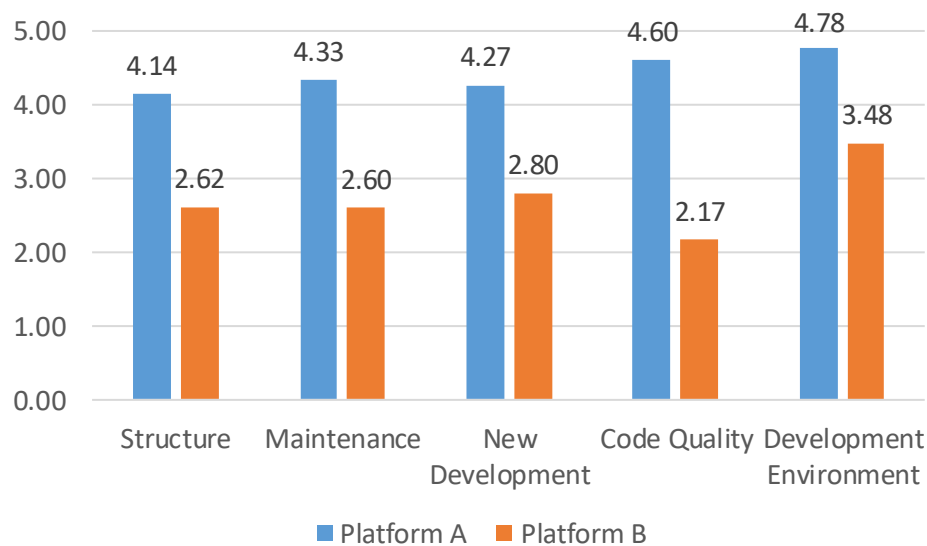


Figure 67. Developer survey section averages

None of the sections on Platform A scored below 4, and all of the sections in Platform B, with the exception of the *Development Environment* section, scored below 3. This indicates a consensus that Platform A performed well in terms of each section evaluated. The *Development Environment* section scored the best for both platforms, due to the fact that these questions evaluated IDEs and third-party libraries. Figure 68 indicates the normal distribution of scores across all answers for each platform.

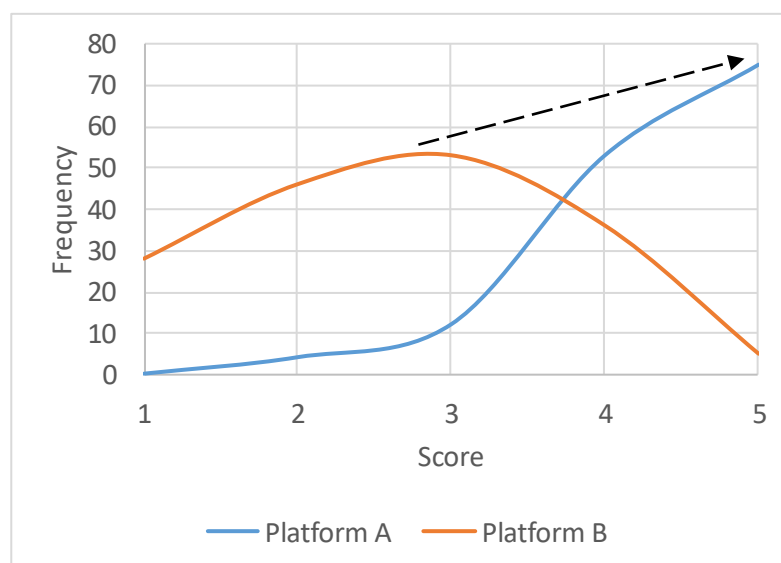


Figure 68. Normal distribution of scores per platform

A clear peak shift of the scores was observed, which further argues that Platform A is perceived as easy to develop for.

Question	Standard Deviation	
	Platform A	Platform B
1	1.000	1.169
2	0.707	0.632
3	0.548	0.632
4	0.548	0.894
5	0.837	0.408
6	1.414	1.378
7	0.548	1.049
8	0.837	1.378
9	0.548	0.753
10	0.548	0.894
11	1.225	1.049
12	0.447	1.049
13	0.548	0.516
14	0.894	0.408
15	0.548	1.211
16	0.447	0.894
17	0.894	0.632
18	0.548	1.329
19	0.447	0.516
20	0.707	1.211
21	0.447	0.816
22	0.447	0.632
23	0.548	0.894
24	0.447	0.408
Average	0.672	0.865

Table 11. Survey questions standard deviation

Table 11 shows the standard deviation of each question. It is noted that the average standard deviation of Platform B is higher than that of Platform A. What follows is a brief discussion on a selection of questions (indicated in yellow in Table 11) for Platform A that yielded anomalous results, or had a higher than average standard deviation.

- **Question 1**

Both platforms have more or less the same learning curve in terms of how long it takes to learn and understand the platform. However, with a high standard deviation for both platforms, this would rather depend on the individual developer and how much past software development experience they have.

- **Question 6**

Both Platform A and B had a high standard deviation. Upon further analysis of scores that were given other than the mode (5 for Platform A, and 4 for Platform B), the only conclusion that could be reached was that the question itself was not perfectly phrased. Respondents assumed the question referred to documentation for the framework architecture itself (as opposed to documentation for components), and not the programming language, such as PHP or C#.

- **Question 11**

One respondent gave a score (2) much lower than the average (4.16) for Platform A. This may be attributed to the fact that the respondent had less than 6 months' experience with Platform A, and only worked on some parts of the platform. Therefore, from their inexperienced point of view, a significant amount of code had to change for architectural bugs.

- **Question 14**

A similar observation was made with question 11, as a respondent gave a score (3) lower than either the average (4.5), or the mode (5). However, the respondent indicated that they have less than 6 months' experience with the platform and only worked with a few parts of the platform during that time. It is therefore understandable that they may find it harder to use existing platform components.

- **Question 17**

The same respondent who caused the anomaly in question 14 indicated a score (3) lower than the average (4.66) and mode (5). The same conclusions are drawn, pointing to inexperience with the platform.

Summary

Platform A, which is based on the framework developed in this study, performed very well according to all respondents who participated in the developer survey discussed in section 4.4. Therefore, the objectives defined in section 1.5 in terms of efficiency, rapid development and acceptance were achieved. A few anomalies were observed in the results, but the issue surrounding these were contributed to inexperienced developers.

4.6. Technical Debt

Another goal defined in section 1.5 is to reduce technical debt:

- *Minimise the accumulation of **technical debt** to lessen the strain on development resources.*

Technical debt is minimised by following the best-practice process illustrated in Figure 36 in section 3.6.2, as well as conforming all coding on the platform to the ESCo’s official coding standard document. A coding standard defines all styles that should be used during development, such as variable and class names, and requiring or forbidding certain practices and methods. The aim of the coding standard is to have all code on the platform formatted in a uniform fashion, which improves code readability and makes maintenance easier.

A third-party tool, SonarQube³⁸, was used to analyse the platform in terms of estimated technical debt. The tool scans through the code and uses a variety of metrics to determine the number of bugs, vulnerabilities, code smells and duplication. An estimated time value for each issue is used to calculate the total amount of time required to address all issues. While this is only a rough estimation, and in the case of some issues an overestimation of the actual time it might take to address the concern, it still provides an indication of how much debt exists in the platform.

The analysis was performed on the platform running on the framework developed in this paper (referred to as “Platform A” for this section), as well as on an existing web platform of the ESCo (referred to as “Platform B”). The results from Platform B will provide a baseline against which to compare the findings. Before an analysis could be performed, SonarQube’s configuration was customised as follows:

- Include only relevant folders with source code in the analysis scope
- Exclude certain third-party libraries’ JS and CSS files from the analysis scope (such as jQuery and Bootstrap)

Table 12 shows a summary of the results from the analysis that was performed on both platforms. A rating is assigned to various metrics that were analysed, such as the platform’s reliability, security and

³⁸ <https://www.sonarqube.org/>

maintainability. Table 13 describes the ratings³⁹. Other metrics, such as the amount of code duplication and number of lines of code, are also summarised.

Platform	Metric				
	Reliability	Security	Maintainability	Duplications	Lines of Code
A	C	A	A	9.2%	86k
B	E	E	A	23.6%	97k

Table 12. Analysis results overview

Rating	Description
A	No issues
B	At least one minor issue
C	At least one major issue
D	At least one critical issue
E	At least one blocker issue

Table 13. Rating descriptions

Platform A achieved a better rating in the reliability and security sections. *Reliability* measures the number of bugs that were found in the code. *Security* indicates how many vulnerabilities due to poor coding practices were found in the platform. *Maintainability* is the total amount of “code smells” present. A code smell is defined as some form of bad-practice code that may indicate a deeper problem in the system. While both platforms had code smells (as shown in Figure 69), no major issues were present that presented a risk to reliability or security. It was also observed that Platform A had a significantly smaller amount of duplicated code. This also contributes to a higher maintainability of the platform, since less code has to change.

Figure 69 shows a more detailed analysis of each platform and compares the metrics from Table 12 for both platforms. The detailed values can be found in Table 16 in Appendix 3.

³⁹ See the SonarQube documentation for more details:
<https://docs.sonarqube.org/display/SONAR/Metric+Definitions>

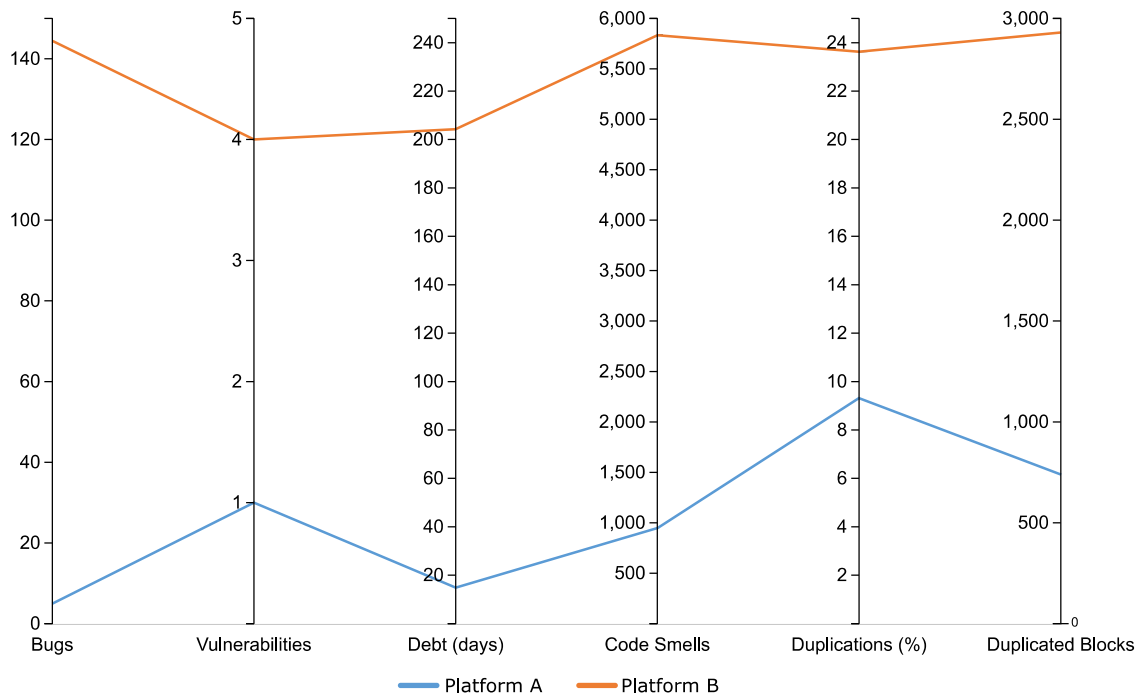


Figure 69. Detailed technical debt analysis to compare platforms

From Table 12 and Figure 69 it is evident that Platform A contains less technical debt across all metrics that were analysed. Figure 70 compares the analysis results from the entire platform against only those of the framework.

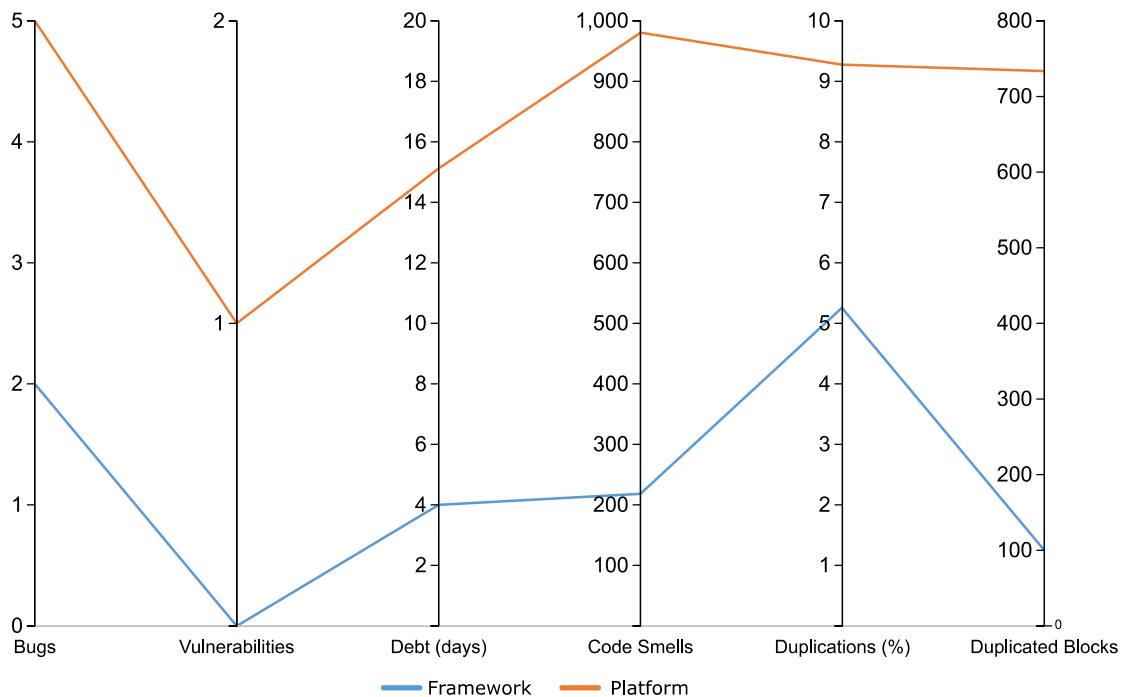


Figure 70. Technical debt analysis of core components vs. entire platform

The detailed results for Figure 70 can be found in Table 17 in Appendix 3. It can be argued that, if the “core” part of the framework (i.e. all the components that are reused by individual web applications) has a very low technical debt, development time and additional technical debt are saved every time that

component is used. If a feature is rewritten from scratch each time it is required, additional technical debt may be incurred.

Summary

From the various measurements and observations made in this subsection, it is clear that the framework (Platform A) has an acceptable amount of technical debt. This validates the objective in section 1.5 in terms of keeping the overall technical debt to a minimum.

4.7. Summary of the Results

Chapter 4 provided validation to ensure that the framework developed in chapter 3 achieved all the objectives and solved the problem stated in chapter 1. Validation was performed via several methods, including an extensive case study, developer survey and technical debt analysis with third-party analysis software.

Table 14 below summarises the methods used to validate each objective stated in section 1.5.

Objective	Validation Method
Modularity	Case study and example functional extension
Efficient maintenance	Developer survey
Minimise technical debt	Technical debt analysis with SonarQube software
Considered as an acceptable framework	Developer survey

Table 14. Validation methods

Chapter 5

Conclusion and Remarks

5.1. Preamble

This chapter provides a summary of the objectives for this study and discusses the results obtained through verification and validation. Shortcomings and scope for future work are also discussed.

5.2. Summary of Work Completed

In the literature study conducted in chapter 2, it was found that the process of developing software may become inefficient due to poor planning or inadequate project specifications. If specifications are changed during later stages of development, a significant amount of development completed may need to change, or even become redundant. Pressure from product delivery deadlines may also cause compromises in quality or functionality in the interest of releasing the software on time.

All these issues lead to the incursion of technical debt in the software. Maintenance is then required in order to pay off the accumulated technical debt. However, maintenance requires the commitment of development resources, which cost time, money and effort that could have been spent on new development.

In addition to maintenance during or after the conclusion of a software project, the deployment and upkeep of software on client devices are also an issue. Special mechanisms and procedures are required to install and keep the software up to date.

The framework developed in this study aims to provide a set of tools and libraries to enable rapid development of web applications. Web applications, or software-as-a-service, provide users with a single point of access to a system, and greatly simplify the maintenance process, since only a single deployment of the software exists.

The aim is to also keep technical debt to a minimum by providing a set of best practices and guidelines in terms of developing new features. A coding standard is also followed to ensure that all code is created according to a unified style, and enforces good programming practices.

An ESCo had the need for a modular framework on which applications and functionality can be rapidly developed. Since their development team consists of only a small number of software engineers, the need for simplified maintenance is crucial.

5.3. Study Conclusion

Prior to developing the framework, a list of requirements and specifications from the ESCo was compiled. These requirements included the connection to an existing database, using and authenticating user accounts from an existing web system, allowing access to a centralised system over the internet and providing access control features.

After the framework was developed, it was verified that each specification was implemented and working according to expectations. In-depth validation was also performed to ensure that the framework solved the problem identified from the literature and satisfied the needs of the ESCo. Validation was performed by multiple means, such as questionnaires, analysis software and examining case studies.

The results yielded the conclusion that the developed framework satisfied all of the ESCo's requirements and achieved all of its objectives. After the development of the framework, several web applications were implemented as case studies by using the available tools and libraries. Analysing the code generated by the case studies showed that significant amounts of code were reused from the framework. If an issue is discovered and fixed in a framework tool or library, that fix will therefore apply to every instance where the tool was used.

Modularity was proven by examining an example where a low-level framework component was changed without it having had an effect on other higher-level components. In this case, if the underlying database technology is changed, the interface between the database and application remains unaffected.

The survey that was conducted evaluated development for the framework in terms of structure, maintainability, new development, code quality and development environment. By comparing the survey results of the framework to those of an existing system of the ESCo, the conclusion was drawn that the framework achieved its goal in terms of enabling efficient development.

Lastly, it was concluded that technical debt in the framework was minimised, as was evident by the technical debt analysis that was performed. It is argued that by eliminating issues from framework components, technical debt is avoided when reusing these components in applications built on the framework.

The framework developed in this study addressed all of the needs and requirements set by the ESCo and identified in the literature. By using existing framework components, best-practice guidelines, and a coding standard, web applications may be developed rapidly and effectively without accumulating a significant amount of technical debt.

5.4. Limitations and Future Scope

While the framework developed in this study was validated to solve the problem that was identified, some aspects with regards to both the design and results may be improved upon.

Firstly, with regards to the validation of the framework, some of the results may be improved upon. A larger sample of developers may be used to complete the development survey in order to evaluate the effectiveness of the framework. While the current sample of seven developers provided a good indication of the effectiveness in the evaluated areas, outliers (due to various reasons) tended to skew the results. More value from the statistical analysis may be obtained from a larger sample. Additionally, some of the questions may be rephrased to better communicate the question or concept being evaluated. The background information section in the survey may also be extended to provide additional information when evaluating an anomalous answer.

By incorporating the best-practice guidelines presented in section 3.6.2 into an SDM (as investigated in section 2.2), further improved efficiency during the development life cycle specifically for this framework could be observed. A more detailed investigation should be done into the methodologies presented, and a recommendation should be made on how to adapt it to include the best-practice guideline. If an SDM takes the process and guidelines into consideration, a multi-disciplinary development team may be able to better coordinate the development of various parts of the web application, such as database transactions, application logic and UI.

Lastly, and on a more technical note, instead of using the ASP.NET Framework, ASP.NET Core⁴⁰ may be used as the base upon which to implement the framework developed in this study. Microsoft provides documentation on the differences between .NET Framework and .NET Core⁴¹, as well as a direct feature comparison⁴². In short, .NET Core features cross-platform compatibility and better performance. Since it is cross-platform, web applications developed on .NET Core may also be deployed on Linux servers

⁴⁰ <https://docs.microsoft.com/en-us/aspnet/core/>

⁴¹ <https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server>

⁴² <https://docs.microsoft.com/en-us/aspnet/core/choose-aspnet-framework>

instead of exclusively on Windows servers. By deploying on a server running a Linux-based operation system, organisations may save licencing costs, since Linux is free to use.

However, a major drawback of .NET Core is that not all functionality offered by .NET Framework is available in .NET Core yet. Similarly, many third-party libraries available through NuGet⁴³ are not yet compatible with .NET Core. Due to this reason, the framework developed in this study is based on .NET Framework in order to make use its full range of functionality offered.

⁴³ See section 3.3.2

Chapter 6

References

- [1] M. Lungu, "Towards reverse engineering software ecosystems," in *2008 IEEE International Conference on Software Maintenance*, 2008, pp. 428–431.
- [2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Softw. Eng.*, vol. 27, no. 1, pp. 1–12, 2001.
- [3] T. Mens, "An Ecosystemic and Socio-Technical View on Software Maintenance and Evolution," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 1–8.
- [4] C.-J. Lin and D.-M. Yeh, "A Software Maintenance Project Size Estimation Tool Based On Cosmic Full Function Point," in *2016 International Computer Symposium (ICS)*, 2016, pp. 555–560.
- [5] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum) - OOPSLA '92*, 1992, pp. 29–30.
- [6] A. Ampatzoglou *et al.*, "The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study," in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, 2016, pp. 9–16.
- [7] E. Parodi, S. Matalonga, D. Macchi, and M. Solari, "Comparing technical debt in student exercises using test driven development, test last and ad hoc programming," in *2016 XLII Latin American Computing Conference (CLEI)*, 2016, pp. 1–10.
- [8] Z. Stachniak, "Early Commercial Electronic Distribution of Software," *IEEE Ann. Hist. Comput.*, vol. 36, no. 1, pp. 39–51, 2014.
- [9] S. Blom, M. Book, V. Gruhn, R. Hrushchak, and A. K, "Write Once, Run Anywhere A Survey of Mobile Runtime Environments," in *2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*, 2008, pp. 132–137.
- [10] S. Geisler, M. Zelazny, S. Christmann, and S. Hagenhoff, "Empirical Analysis of Usage and Acceptance of Software Distribution Methods on Mobile Devices," in *2011 10th International Conference on Mobile Business*, 2011, pp. 210–218.
- [11] M. Dillinger and R. Becher, "Decentralized software distribution for SDR terminals," *IEEE Wirel. Commun.*, vol. 9, no. 2, pp. 20–25, Apr. 2002.
- [12] P. K. Chouhan, F. Yao, and S. Sezer, "Software as a service: Understanding security issues," in *2015 Science and Information Conference (SAI)*, 2015, pp. 162–170.
- [13] M. V. Mantyla and J. Vanhanen, "Software Deployment Activities and Challenges - A Case Study of Four Software Product Companies," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 131–140.
- [14] V. B. S. Silva, F. Schramm, and A. C. Damasceno, "A multicriteria approach for selection of agile methodologies in software development projects," in *2016 IEEE International Conference on*

- Systems, Man, and Cybernetics (SMC)*, 2016, pp. 002056–002060.
- [15] C. N. Ojeda-Guerra, “A Simple Software Development Methodology Based on MVP for Android Applications in a Classroom Context,” in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 2015, pp. 1429–1434.
 - [16] S. M. Mitchell and C. B. Seaman, “A comparison of software cost, duration, and quality for waterfall vs. iterative and incremental development: A systematic review,” in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 511–515.
 - [17] A. F. G. Filho *et al.*, “Agile Software Development Learning through Open Hardware Project,” in *2015 6th Brazilian Workshop on Agile Methods (WBMA)*, 2015, pp. 40–47.
 - [18] M. M. Jha, R. M. F. Vilardell, and J. Narayan, “Scaling Agile Scrum Software Development: Providing Agility and Quality to Platform Development by Reducing Time to Market,” in *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*, 2016, pp. 84–88.
 - [19] N. M. N. Daud, N. A. A. A. Bakar, and H. M. Rusli, “Implementing rapid application development (RAD) methodology in developing practical training application system,” in *2010 International Symposium on Information Technology*, 2010, pp. 1664–1667.
 - [20] J. Martin, *Rapid Application Development*. Macmillan Coll Div, 1991.
 - [21] Y. Khmelevsky, X. Li, and S. Madnick, “Software development using agile and scrum in distributed teams,” in *2017 Annual IEEE International Systems Conference (SysCon)*, 2017, pp. 1–4.
 - [22] W. Singhto and N. Phakdee, “Adopting a combination of Scrum and Waterfall methodologies in developing Tailor-made SaaS products for Thai Service and manufacturing SMEs,” in *2016 International Computer Science and Engineering Conference (ICSEC)*, 2016, pp. 1–6.
 - [23] S. Hassan, U. Qamar, and M. A. Idris, “Purification of requirement engineering model for rapid application development,” in *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2015, pp. 357–362.
 - [24] B. Prashanth Kumar and Y. Prashanth, “Improving the Rapid Application Development process model,” in *2014 Conference on IT in Business, Industry and Government (CSIBIG)*, 2014, pp. 1–3.
 - [25] V. Lenarduzzi, A. Sillitti, and D. Taibi, “Analyzing Forty Years of Software Maintenance Models,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 146–148.
 - [26] K. Srewuttanapitikul and P. Muengchaisri, “Prioritizing software maintenance plan by analyzing user feedback,” in *2016 International Conference on Information Science and Security (ICISS)*, 2016, pp. 1–5.
 - [27] R. E. Fairley and M. J. Willshire, “Better Now Than Later: Managing Technical Debt in Systems

- Development,” *Computer (Long. Beach. Calif.)*, vol. 50, no. 5, pp. 80–87, May 2017.
- [28] C. Izurieta, G. Rojas, and I. Griffith, “Preemptive Management of Model Driven Technical Debt for Improving Software Quality,” in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA '15*, 2015, pp. 31–36.
 - [29] S. H. Vathsavayi and K. Systa, “Technical Debt Management with Genetic Algorithms,” in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2016, pp. 50–53.
 - [30] M. Soltanian, R. B. S. Mysore, and B. Ottersten, “Reliability problems and Pareto-optimality in cognitive radar (Invited paper),” in *2016 24th European Signal Processing Conference (EUSIPCO)*, 2016, pp. 2225–2229.
 - [31] N. Ramasubbu, C. F. Kemerer, and C. J. Woodard, “Managing Technical Debt: Insights from Recent Empirical Evidence,” *IEEE Softw.*, vol. 32, no. 2, pp. 22–25, Mar. 2015.
 - [32] A. Y. Gital *et al.*, “Performance analysis of cloud-based CVE communication architecture in comparison with the traditional client server, P2P and hybrid models,” in *The 5th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2014, pp. 1–6.
 - [33] E. Borowsky, A. Logan, and R. Signorile, “Leveraging the Client-Server Model in P2P: Managing Concurrent File Updates in a P2P System,” in *Advanced Int’l Conference on Telecommunications and Int’l Conference on Internet and Web Applications and Services (AICT-ICIW’06)*, 2006, pp. 114–114.
 - [34] W. Kiess, X. An, and S. Beker, “Software-as-a-Service for the Virtualization of Mobile Network Gateways,” in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–6.
 - [35] M. A. Constanzo and S. Casas, “Usability evaluation of web support frameworks,” in *2016 XLII Latin American Computing Conference (CLEI)*, 2016, pp. 1–6.
 - [36] M. R. Islam, M. R. Islam, M. M. Islam, and T. Halim, “A study of code cloning in server pages of web applications developed using classic ASP.NET and ASP.NET MVC framework,” in *14th International Conference on Computer and Information Technology (ICCIT 2011)*, 2011, pp. 497–502.
 - [37] A. B. Mnaoue, A. Shekhar, and Zhao Yi Liang, “A generic framework for rapid application development of mobile web services with dynamic workflow management,” in *IEEE International Conference on Services Computing, 2004. (SCC 2004). Proceedings. 2004*, pp. 165–171.
 - [38] H. Fleischmann, J. Kohl, and J. Franke, “A modular web framework for socio-CPS-based condition monitoring,” in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, 2016, pp. 1–8.
 - [39] W. Naruephiphat, R. Prom-Ya, and C. Chansripinyo, “A web-based management system design

- for wireless sensor network monitoring,” in *2013 International Computer Science and Engineering Conference (ICSEC)*, 2013, pp. 281–285.
- [40] P. Worrall and T. Chausalet, “Development of a web-based system using the model view controller paradigm to facilitate regional long-term care planning,” in *2011 24th International Symposium on Computer-Based Medical Systems (CBMS)*, 2011, pp. 1–7.
- [41] M. Jailia, A. Kumar, M. Agarwal, and I. Sinha, “Behavior of MVC (Model View Controller) based Web Application developed in PHP and .NET framework,” in *2016 International Conference on ICT in Business Industry & Government (ICTBIG)*, 2016, pp. 1–5.
- [42] J. Oh, W. H. Ahn, and T. Kim, “MVC architecture driven restructuring to achieve client-side web page composition,” in *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2016, pp. 45–53.
- [43] X. Li and N. Liu, “Research on L-MVC Framework,” in *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2016, pp. 151–154.
- [44] C. Zhou and Q. Fei, “Warehouse Management System Development Base on Open Source Web Framework,” in *2016 International Conference on Industrial Informatics - Computing Technology, Intelligent Technology, Industrial Information Integration (ICIICII)*, 2016, pp. 65–68.

Chapter 7

Appendices

Appendix 1

This appendix contains a more detailed overview of modular components that were developed for use within the framework.

Grid

The grid component is a jQuery plugin that provides a customisable grid element. It will automatically bind to a set of Models and provide full CRUD functionality. The grid is highly configurable in order to suit a wide variety of requirements. Some of the functionality includes:

- Customisable controls
- Column dependencies
- Searchable content
- Sortable columns
- Paginated display
- Conditional actions
- Custom styling

The grid integrates with the MVC architecture by rendering directly into a Partial View. A Partial View is a special type of view component that may be dynamically rendered into an existing View. The grid's Partial View is bound to a View Model, which contains a collection of Models. A row is added for each Model, and properties automatically assigned and bound to cells in the row.

Linking Table

This jQuery plugin provides a generic user interface to manage associative tables⁴⁴. The component may be customised in its appearance to suit the needs of the application.

Tree

This jQuery plugin displays an interactive tree view of hierarchical objects. Each node in the tree view may be expanded or collapsed.

⁴⁴ A database table that maintains many-to-many relationships. This means that a table contains at least two columns where entries from other tables are referenced and associated with each other.

Tabs

The tabs component is a jQuery plugin used to toggle the visibility of various elements on the user interface. Each tab is linked to a specific element, and by selecting a tab, only that tab's element will be visible.

Controls

The controls jQuery plugin is simply used to provide a set of configurable control buttons, and each button may be configured to execute a specific function. This component also ensures that all controls are rendered consistently across the entire platform.

JS Utilities

An additional set of JS utilities is available, providing functions to control various common page elements. These functions include the following:

- Show or hide a loading bar
- Show or hide a sidebar panel
- Display a message to the user

Filters

A Filter is a concept in ASP.NET that allows a developer to define specific functionality that should be executed when a user request is directed to a Controller's action method. The filter can be configured to either execute prior to or after executing the action method. Once a filter has been created, it may either be attached to individual Controller actions, the Controller itself (to include it to all actions) or globally. Global filters will be attached to all actions in the ASP.NET application.

In the scope of the framework, the following filters were created:

- **Login Filter**

Provides user access control. The filter checks if a user is logged in to the current session, and if so, will allow access to the requested resource. This filter is attached at a global level, and may selectively be disabled for certain actions.

- **External Login Filter**

Provides user access control for external systems and requests. The filter controls access to resources from external systems. Login credentials are sent along with the request, and the filter then authenticates these to ensure that a valid user account is being used to access the platform.

- **Area Access**

Provides user access control to specified areas in the platform. The filter checks if a specified user access right has been granted to the logged-in user, and if so, will allow the user to request pages from that area.

- **Menu Access**

Provides user access control to specific pages or menus. The filter checks if a user has been granted access to a specific page, and if so, will allow the user to access that page. This filter differs from “Access Area” in the sense that “Access Area” controls an entire area, and “Menu Access” can further narrow down access right to specific pages within an area.

Library

The library contains a collection of common classes and functionality that is generally available for use in the entire platform. Library classes include the following:

- Email functionality
- Class with constant definitions, such as date formats and folder locations
- Encryption functionality
- Event logging
- Random value generator
- Menu registration
- Password manager

Each application implemented on the framework may also have its own library that contains functionality specific to that application.

Graphics

All graphical assets, such as images and icons, are centralised in a single folder. This makes it easier for a developer to locate or reuse a specific asset.

Appendix 2

Table 15 below contains the raw results recorded from the developer survey forms.

		Question																											
Platform	Respondent	A	B	C	D	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
A	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	2	b	b	a	a	4	3	5	5	5	5	5	4	4	5	5	4	4	4	5	5	5	5	5	4	5	5	5	5
	3	a	a	b	a	2	4	4	4	4	3	4	5	4	5	5	3	3	5	4	5	5	4	5	4	5	5	4	5
	4	b	a	a	a	2	5	4	4	4	2	4	5	5	5	4	4	3	5	4	5	5	5	5	5	5	5	5	5
	5	a	b	a	a	4	4	5	5	5	5	5	3	5	4	2	4	4	5	5	5	5	5	5	4	5	5	5	5
	6	a	c	a	a	3	4	4	4	3	5	4	4	4	4	4	4	3	3	4	4	3	4	4	3	4	4	4	4
	7	a	b	a	a	5	5	5	5	4	5	4	4	4	4	5	5	5	4	5	5	5	5	5	4	5	5	5	5
	Average					3	4	5	5	4	4	4	4	4	4	5	4	4	4	5	5	5	5	5	5	4	5	5	5
B	1	e	a	b	b	3	2	3	3	1	4	4	3	3	4	2	3	3	3	3	2	2	3	2	4	3	3	4	4
	2	c	a	c	b	2	1	3	2	1	4	2	1	2	3	4	4	3	3	4	3	3	1	2	3	3	3	4	4
	3	a	b	c	b	5	2	3	2	1	3	3	1	2	2	2	5	4	2	1	1	2	1	1	2	2	2	3	4
	4	b	b	a	a	4	3	4	4	1	1	3	4	3	3	1	4	4	3	2	2	2	3	2	2	2	3	5	4
	5	a	b	c	a	2	2	2	3	1	5	1	2	1	2	3	2	3	3	2	1	1	1	1	1	1	3	3	3
	6	a	c	a	a	3	2	3	4	2	4	2	4	2	4	3	3	3	3	4	3	2	4	2	4	3	4	5	4
	7	e	a	b	b	3	2	3	2	1	4	4	3	4	2	2	3	2	4	2	1	3	2	1	1	4	3	1	4
	Average					3	2	3	3	1	4	3	3	2	3	2	3	3	3	3	2	2	2	2	2	3	3	4	4

Table 15. Developer survey results

Appendix 3

This appendix contains tables with detailed values of the technical debt analysis performed in section 4.6.

Category	Platform	
	A	B
Bugs	5	144
Vulnerabilities	1	4
Debt	15 days	206 days
Code Smells	974	5800
Duplications	9.2%	23.6%
Duplicated Blocks	736	2900

Table 16. Detailed analysis results

Category	Code	
	Framework	Platform
Bugs	2	5
Vulnerabilities	0	1
Debt	4 days	15 days
Code Smells	215	974
Duplications	5.2	9.2
Duplicated Blocks	104	736

Table 17. Framework vs. platform technical debt analysis