



CREATING AND AUTOMATING BROWSER BASED TESTS FOR A WEB-APPLICATION IN MAINTENANCE

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering and Digital Transformation, Master's thesis

2023

Ville Martas

Examiners: Associate Professor Jussi Kasurinen

Assistant Professor Dominik Siemon

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Ville Martas

CREATING AND AUTOMATING BROWSER BASED TESTS FOR A WEB-APPLICATION IN MAINTENANCE

Master's thesis

2023

49 pages, 10 figures, 1 table

Examiners: Associate Professor Jussi Kasurinen and Assistant Professor Dominik Siemon

Keywords: software testing, test automation, DevOps, CI/CD pipeline

Automation of the development process is essential for modern software projects. Deploying a software application usually requires the same predictable actions and introducing automation to a project could reduce the time between deploying new versions and decrease the number inevitable human errors that happen because of manual processes. As software projects grow and maintaining gets more difficult, the importance of software testing also increases. Testing all the different views and functionality of an application that has been developed for over two decades would take a lot of time to test manually, although creating automated tests for all the features would also require a considerable investment.

This thesis investigates if value can be gained from introducing development process automation and testing to a software project that is already in the maintenance phase. Selenium Webdriver is used for creating a browser automation test suite prototype. The test suite is later automated to the development process of the application by using Azure DevOps Pipelines services. Results show that introducing automation to the project offers noticeable improvements in the development process and that a test framework can be created for an existing project. The challenges that were faced were regarding the end-to-end test cases themselves. Tests running on the browser were occasionally failing because of reasons not directly related to the functionality of the application under testing, which produced unreliable test results. Getting the tests from failing back to passing would often require changes to the tests themselves or the environment where the application is running, not to the application under testing itself.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Ville Martas

SELAINPOHJAISTEN TESTIEN LUONTI JA AUTOMATISOINTI WEB-SOVELLUKSELLE YLLÄPIDOSSA

Tietotekniikan diplomityö

2023

49 sivua, 10 kuvaa, 1 taulukko

Tarkastaja: Apulaisprofessori Jussi Kasurinen ja Apulaisprofessori Dominik Siemon

Avainsanat: ohjelmistotestaus, testiautomaatio, DevOps, CI/CD pipeline

Kehitysprosessin automatisointi on olennainen osa moderneja ohjelmistoprojekteja. Sovelluksen julkaisu vaatii yleensä samat ennustettavat toimenpiteet, ja automaation käyttöönotto projektiin voi lyhentää uusien versioiden julkaisuväliä ja vähentää manuaalisista prosesseista aiheutuvien väistämättömien inhimillisten virheiden määrää. Ohjelmistoprojektien kasvaessa myös ylläpito vaikeutuu, ja ohjelmistotestauksen merkitys korostuu. Kaikkien eri näkymien ja toiminnallisuuksien testaaminen sovelluksessa, jota on kehitetty yli kahden vuosikymmenen ajan veisi paljon aikaa testata manuaalisesti, vaikkakin automatisoitujen testien luominen kaikille ominaisuuksille vaatisi myös merkittäviä investointeja.

Tässä lopputyössä selvitetään, voidaanko lisäarvoa saada kehitysprosessin automatisoinnista ja automatisoiduista testeistä, kun kyseessä on jo ylläpitovaiheessa oleva ohjelmistoprojekti. Selenium Webdriveria käytetään selainpohjaisen testiprojektin prototyypin luomiseen. Testisarja automatisoidaan myöhemmin sovelluksen kehitysprosessiin Azure DevOps Pipelines -palveluiden avulla. Tulokset osoittavat, että automaation käyttöönotto projektiin parantaa kehitysprosessissa ja että toimivat testiprojekti voidaan luoda ylläpidossa olevalle ohjelmistoprojektille. Kohdatut haasteet koskivat pääasiassa itse testitapauksia. Selaimessa suoritettavat testit epäonnistuivat ajoittain syistä, jotka eivät olleet suoraan yhteydessä testattavan sovelluksen toiminnallisuuteen, mikä tuotti epäluotettavia testituloksia. Testien havaitsemien ongelmien korjaus vaati usein muutoksia itse testeihin tai ympäristöön, jossa sovellus oli julkaistu eikä itse sovellukseen johon testaus kohdistui.

ACKNOWLEDGEMENTS

Thank you to my family, friends, supervisor Jussi Kasurinen, Arto and rest of the team at work for their support throughout the diploma-related development and academic work.

ABBREVIATIONS

DevOps	Development Operations
RQ	Research question
SQL	Structured Query Language
SSMS	SQL Server Management Studio
SSRS	SQL Server Reporting Service
UI	User interface
E2E	End-to-end

Table of contents

Abstract

Acknowledgements

Abbreviations

1	Introduction	8
1.1	Goals and delimitation	8
1.2	Practical implementation.....	9
1.3	Thesis structure	10
2	Research questions and research method	11
2.1	Research questions	11
2.2	Research method: case study	12
2.2.1	Preparation	13
2.2.2	Collecting evidence.....	13
2.2.3	Reporting	13
3	Related research: software testing and test automation.....	14
3.1	Software testing importance and terminology	14
3.2	End-to-end testing	15
3.3	Test automation.....	17
3.4	DevOps, CI/CD and testing.....	18
4	Technical specifications and context of the application under testing	19
4.1	Application under testing	19
4.2	Application context	20
5	Test framework.....	22
5.1	Test project with Selenium.....	22
5.1.1	Page Object Model pattern	23
5.2	Test cases	25
5.3	Challenges in testing	27
5.3.1	Test data generation	28
6	Configuring the CI/CD pipeline in Azure DevOps environment.....	30

6.1	Azure DevOps workflow	30
6.2	Azure Pipelines	31
6.2.1	Build pipeline.....	32
6.2.2	Release pipeline	36
6.3	Release pipeline test results	37
7	Results and future development	39
7.1	Limitations and validity	41
8	Discussion.....	42
9	Conclusions	46
	References.....	47

1 Introduction

Software projects that have been in development and later in maintenance for multiple years get gradually more complex over time. Projects have often been worked by multiple people over the years and different approaches by developers to problems inevitably lead to technical debt which complicates and slows down the development later down the line. To keep the development at these later stages manageable, projects can greatly benefit from having a well-structured development process. The term DevOps, abbreviation of development operations, has been very popular within the software industry over the recent years. Ebert et al. describe DevOps as the actions of integrating the development and the operations of a project together by automating development, application deployment and the monitoring services of a project [10]. As automation is introduced to the development process of an application, automated tests also become a possibility. While testing can considerably affect the quality of the software, automating the tests can reduce the project costs considerably as manual work from repetitive tasks decreases as automation increases. With an automated testing framework problems with software can be discovered before new versions of an application get deployed from the development environment to staging environment and later to production. This can affect project costs since publishing a new build to the production environment can be expensive and time consuming if unnoticed problems often get caught too late and require additional maintenance work and deployment meetings that could have been avoided. Automated testing does however require an investment since designing and creating a test framework and automating it within the development process does require additional work, which could be spent on developing and maintaining the application itself. Later additional costs also come from maintaining the test framework since for the system to be useful, the tests need to be kept updated for the test results to be useful.

1.1 Goals and delimitations

In this thesis a case study is conducted on how running a test suite could be automated within the development process of an existing software application that has been in development

and in maintenance for over two decades. The first goal of the thesis is to analyse how creating a prototype test project that targets the functionality of an already built and deployed version of the application in a production-like environment can be achieved. The tests created are end-to-end (E2E) tests that simulate real user scenarios of using the application. The tests should be as close to the typical routines that a developer or a tester from the user's side would manually test on the browser when changes to the application occur. The research focus is on the test design to keep the test project maintainable if scaled over time. Test results are also a point of interest in the study, since the "flaky" nature of E2E tests may be an even larger concern with an application running on older technology. Flakiness in E2E testing context means that even if the functionality in the application is correct, the test results can vary between test runs because of for example network, service, or performance related problems.

The second goal is to automate the created test project within the deployment pipeline, so the tests automatically run on new versions of the application. This would offer the development team information about a new version automatically and relatively fast without having to test basic functionality manually. Automation does however require resources and time for first setting the system up and later maintaining it if the application under testing changes. Azure DevOps services Pipelines and Releases are used for the automation parts of the development.

The development focuses on initially creating the test project and later working with services that handle the automation of the development and deployment process. Functional changes to the application under testing are avoided but small necessary configuration related changes may be required.

1.2 Practical implementation

The practical development for the thesis is a part of a commission work that was proposed to the customer regarding a application that is developed and maintained by Evitec, a company offering software solutions and consulting services for the finance sector, and analytics and data management services for various other sectors [11]. The software project has initially some automation within the development process regarding building the source code of the application and creating a build package that can be deployed to the local

development environment. Pipelines and Releases from the Azure DevOps [4] services are used for the automation of the development process. Additional work to the automation process is required to first include the test project to the building part of the pipeline, and later to add and run the tests automatically on the development environment if the deployment was a success. For the testing, different end-to-end testing tools will be considered such as Selenium [22] and Cypress [16] from which the most suitable one is chosen to be used for building a prototype project. The test cases should be the most common routines that a tester from the development team or from the customer's side would try to manually do when a new build is being deployed to the customer's test environment and later to production. Individual test cases are discussed and defined together with the development team and the client.

1.3 Thesis structure

The thesis is divided into nine chapters. The first chapter is an introduction to the topics, the goals and delimitations and the practical implementation regarding the thesis. The second chapter lists and explains the research questions which the project is set out to answer and research methods used within the project. The third chapter focuses on related research regarding software testing, automation of a software development process, and software test automation. In chapter four the application under testing is introduced by giving information on a technical level and explaining the basic concepts that the users of the application would interact with. Fifth chapter focuses on the technical implementation of the test framework that was created with Selenium. The architecture of the framework as well as the individual test cases is discussed on this chapter. Chapter six focuses on the automation part of the project by first explaining the typical workflow in the Azure DevOps environment and how the automation was introduced to improve the development process. The seventh chapter focuses on the results found regarding the development of the automation, the test project and later combining the two and limitations and validity of the research. In chapter eight the findings are discussed, and the research questions are answered according to the findings. The thesis ends with a conclusion where the process and the findings are concluded.

2 Research questions and research method

This chapter defines the research questions, and the research method of case study. Case study is first introduced generally as a research method, then by how it is specifically being applied within this thesis.

2.1 Research questions

The research questions of the thesis are introduced by first stating the research question (RQ), and then giving an explanation on why the specific question is important to this study. Research questions RQ1 and RQ2 are the main research questions of the study, while RQ3 and RQ4 support the findings for the first two questions.

RQ1: Can a testing framework be built for an existing software application to offer an additional layer of validation to the development process even if introduced later in the software development lifecycle?

Developing a test framework and automating the development process is preferably and usually started at the beginning of a software project. If the project has gone through the development phase and has been in maintenance for multiple years, can the test framework still be built to help ensure quality and reliability within the project?

RQ2: Can testing and test automation introduced late in the software development life cycle still offer value to a software project and how could the most value be achieved when designing the test cases?

If the test framework can be built and incorporated to the deployment pipeline, can this still offer value to the software project. Can we see valuable test results directly after only a few tests that have been automated or would this require extensive test development to meet all the hundreds if not thousands of testable features that have been built over the years. How the test cases are selected may play a role in the value question as well: should the new test cases target the functionality that is currently worked on, or should the focus be on the most used and common features even if current maintenance work does not concern these parts of the application?

RQ3: What problems can arise when adding development process automation on an application that has been in maintenance for multiple years?

Some of the tools used to build the application initially have lost official support during the years, and migrations to new technology have been made during the development and maintenance periods. Using the modern automation tools such as Azure DevOps Pipelines may conflict with the older web-application.

RQ4: What are the typical problems of automated E2E tests in a software project.

End to end testing can offer the advantages of testing functional aspects of the application by running similar routines that a normal user would. E2E tests however have the tendency of being “flaky”, meaning that since so many different things contribute to a test either passing or failing, false positives or false negatives may happen. E2E tests can also take a lot more time to run, so immediate feedback regarding the results may not be possible.

2.2 Research method: case study

The research method used in this thesis is a case study. Case study as a research method is about focusing on a case or cases within a real-world context. In “Checklist for Software Engineering Case Study Research” Host and Runeson propose a checklist that consists of 38 subtasks for researchers conducting software engineering related case studies. According to the paper, a typical process for a case study can be divided into five separate phases: case study design, preparation for data collection, collecting evidence, analysis of the collected data and reporting. The individual subtasks highlight and remind of important parts of the case study process that a researcher or researchers should keep in mind. [15, 31]

The objective of this case study is to answer the research questions defined by collecting information from performing a work assignment that was defined together with the software product owner. Concepts from [15] are used to define the process for this specific case study. The process can be divided into three stages: preparation, collecting evidence, and reporting.

2.2.1 Preparation

At the preparation stage, information is gathered regarding E2E testing, test automation and automating the development process from research done previously. If other similar cases are found where test automation is added later in the software lifecycle these are also studied in preparation. Practical information and tips are also gathered from senior members from the company who have prior experience from working with test automation, end-to-end tests and especially the Azure DevOps environment. Some of the individual test cases are also defined together with the development team before the actual development starts.

2.2.2 Collecting evidence

Collecting the evidence starts by first focusing on building a test framework with only a few test cases. The goal is to prove that tests running in the browser can be created for the application and that the individual tests run as expected simulating how a real user would use the application. The second part of evidence collecting is to demonstrate how the test framework can be set up to automatically run within the deployment pipeline when new changes get pushed to the source control. If these parts can be successfully completed, the focus is on the design of the test framework to keep it maintainable, and understanding what type of real-world routines in the application should and should not be turned into test cases. Evidence is also collected from monitoring the test case results and discussing with the development team on what possibilities or others see for further development.

2.2.3 Reporting

In the reporting stage the findings are analysed and discussed based on what was done during the work assignment. The proposed research questions will be answered based on the succession of the initial plan and the development done during the evidence collection. Observations and possible problems will be discussed after the development has concluded.

3 Related research: software testing and test automation

This chapter discusses related research regarding software testing and software test automation. What is the role of software testing within software products lifecycle, and why testing is important for software projects. E2E testing is also discussed more in-depth to better understand the types of tests that are created in this project. It is valuable to understand how testing frameworks have been built in the past and what are the common problems that arise when the frameworks have been designed and built.

3.1 Software testing importance and terminology

The goal of software testing is to identify faults within an application by executing a part of the application under testing and comparing the outputs to predetermined expected values. Software testing can never verify that the software does not have any faults, testing can only notify of existing problems that happen to get caught by the tests. Testing is an extremely important factor regarding the quality of a software project. Ideally the people practicing the testing activities in a project would be different from the people who build the software. [23]

Software testing terminology includes terms such as a test, test case and a test suite. Test and test case according to can be used as synonyms. A test case or a test is performing a set of actions to a system and determining if the system satisfies the requirements as expected [29]. A Test suite is a collection of test cases [23].

Errors, faults, and flaws in software are often referred generally as “bugs” in software development. In “Introduction to software testing”, Ammann and Offutt introduce definitions for software faults, software errors and software failures. A Software fault is described as “A static defect in the software”, meaning that the software basically functions incorrectly because of a fault in the design and not in the code itself. A software fault can lead into a software error which is the result of the internal state of the application being incorrect due to the faults in design. Software failure is described as incorrect behaviour where the software does not function as expected by the requirements. In “Software testing” [23] software failure is described as “dynamic in nature”, meaning that failures relate to problems observed during the execution of the application. Failures occur because of the

faults in the design leading into errors which then fail the execution leading to software failure. [1]

Software testing is a very expensive and time-consuming part of software development. According to Singh's book "Software testing" [23], software testing can take from one third to half the costs of the entire software project. In a research article "Software Test Automation in Practice: Empirical Observations", Kasurinen et al. gathered and analysed data from multiple large and medium sized companies and organizations regarding their test automation processes. Findings showed that the development effort spent in testing by the companies was considerably lower than other literature suggested, median 25% development effort compared to 50% - 60% that was expected. Resources spent on the automation part of testing was also lower than expected. Interviews with the companies showed that the size of the project and the amount of resources was a key factor on whether automation was thought as something worthwhile to invest into. Generally, if the resources were limited in the project, test automation was considered to be not worth the investment. [17]

There are several different types of software testing, such as unit testing, integration testing, functional testing, and end-to-end testing. With unit testing the goal is to test low level components of the software by running small piece of code, for example a single method in isolation and verifying that they produce the correct result with the given inputs. Integration tests focus on testing multiple lower-level components of the software to see that the components work together. Functional tests are often business requirement related, and the focus is on getting the correct product specific results. End-to-end (E2E) tests that are the focus on this research, focus on determining that the application works as intended in simulated real life situations from the users perspective. [25, 27]

3.2 End-to-end testing

End to end testing is about testing the functionality of the application by focusing on the end-user's perspective with test cases. An end-to-end test case can be for example the whole process of simulating an user logging in to an application, doing a set of actions that an application user would do and verifying that the entire flow from start to finish works as intended [25]. The scenarios created usually relate to the actual use cases of the application

[26]. Determining if the functionality that is tested works as intended or not with E2E tests can be a challenge. When testing for example a function or a component in a software project on a lower level with unit tests, it's often clear if the software component functions as expected by the test results. With end-to-end testing it is stressed that tests only notify of the existing problems as stated in Singh's study [23] regarding software testing.

End-to-end tests have a tendency of being "flaky" meaning that consecutive runs of the same tests can sometimes lead to tests passing and tests fails without a clear reason [19]. Since the system is tested on a production like environment, a single test case being successful also requires everything to work in the environment and around the application as well. If the application is running slower because the server is under a lot of stress, or there are connection related problems that are not caused by something wrong with the application, the tests can also be affected. Some asynchronous actions might require the browser driver to wait for the sites action to finish before continuing with the test actions or validating expected conditions. Olinas et al. researched the flakiness regarding end-to-end test suites in "Reducing Flakiness in End-to-End Test Suites: An Experience Report" [21]. In the report the test suite that was built tried to solve some of the problems with flakiness by adding thread sleeps to the test specific code and the page related actions. Thread sleeps would pause the test actions for a specified amount of time for the application to manage for example asynchronous calls that require time to finish. This was deemed as not an ideal way to handle asynchronous calls since the thread sleeps themselves had been noticed to cause even more flakiness as they affected the asynchronous calls in negative ways. Stopping the execution for a fixed amount of time also led to the tests taking longer to run. [21]

Leotta et al. studied the challenges of End-to-End testing with Selenium WebDriver [19]. Some of the challenges proposed included slowness of running the tests, fragility of the tests, the flakiness, difficulties regarding the maintainability of the tests during development, time it takes to develop the tests, and difficulties regarding the failure analysis of the test cases e.g., difficulties with troubleshooting the reason a test failed in the first place. Based on the study surveying 78 experienced participants, most challenges were faced with tests interacting with asynchronous content in the web pages and the brittleness also fragility of the tests themselves. [19]

3.3 Test automation

Since the productivity of the development has increased over the years considerably as the tools have gotten better this has also led to more features over time requiring testing faster than before [28]. To be efficient with testing the tests should be run often, for example after every time a new change is made to the main development branch or regularly overnight. Automated tests can be very expensive depending on the type of tests that are automated. Running a test suite of unit tests within the build pipeline can be a very fast and inexpensive task for the pipeline to run. Automated E2E tests however require the build pipeline to first be ran to produce a build of the application, and then deploy it on a development environment where the tests can then be run. After the environment is ready, the tests themselves simulate real users navigating through the application, and a single test can take over a minute to finish running. If the plan is to create a comprehensive test suite, this can lead to the test suite running for a very long time if there is no parallel test running available. Vila et al. mention in [28] that not everything can be automated, and manual testing can't be completely replaced with automated tests since the succession of some tests require human intervention and manual work. Vila et al. mention that important things to consider when automation testing is considered are the independency of test cases, scalability of the test framework and test project is separated from the application under testing [28].

Christophe et al. studied in [6] how Selenium tests are used and maintained within open source projects. While automating the test execution does allow the tests to run repeatedly during development this does introduce the problem of maintaining the test scripts themselves according to the research. The research discusses the fact that as the application under testing evolves and grows, tests are bound to break over time and then require additional maintenance resources which costs companies a lot of money. Example from [20] was used: 74% of tests scripts of Adobe Acrobat Reader break during two successive releases. One of the research questions in [6] was regarding the co-evolving of the Selenium test scripts together with the underlying application. Data from a total of 287 open-source Java-based web application projects using Selenium was gathered and results suggested that on average it takes 11.23 commits, or 4.33 days before a Selenium test script is affected by a commit, suggesting that Selenium does evolve as the project evolves. [6]

3.4 DevOps, CI/CD and testing

In [18] Lai et al. researched improving the efficiency of software maintenance of a banking service by introducing CI/CD pipeline workflow. Five different phases for an effective CI/CD pipeline phases are laid out: build phase, unit testing phase, deployment phase, automated test phase and product deployment phase. Quality of the CI/CD pipeline according to the conference paper was directly related to the efficiency and success of development operations culture that modern software development companies exercise.

In [8] Cruzes et al. aim to understand how DevOps affects modern software testing principles, what types of approaches there are for testing within DevOps and what challenges can occur. Practices of an IT consultancy firm from Norway was investigated by interviewing testing related personnel within the organisation. Interviews pointed out that testing in DevOps environment often gets so incorporated to the development itself, that a clear understanding of what is considered testing and what development can be very difficult. Emphasis by the testers was put on the usefulness of the automated tests, but also to the fact that continuous testing does not equal test automation, but that manual testing is also very important within continuous testing of a DevOps environment. Maturity of the DevOps development chain and getting the whole development team to work with same quality standards were mentioned as challenges within DevOps testing. [8]

Claps et al. studied the technical and social challenges of continuous deployment phenomenon in software industry in “On the journey to continuous deployment: Technical and social challenges along the way” [7]. Atlassian was the target company of the case study. Continuous deployment of the application was generally seen as a more difficult task compared to continuous integration, since the product the company offered consisted of multiple coupled applications which add to the complexity of the deployment process. Technical challenges faced withing continuous delivery included challenges regarding testing, product quality, source control, database management and continuous integration. Social challenges included things like coordination in and across teams, experience of the team adopting CD practices, and customers not noticing new features added to the product. [7]

4 Technical specifications and context of the application under testing

This chapter contains background information about the application for which the automated testing framework is being developed for. Technical details are discussed as well as the business-related background regarding the application domain.

4.1 Application under testing

The underlying financing related application was initially built with Visual Basic 6 (VB6) programming language, also called Classic Visual Basic. Later the application was converted to VB.NET as official support for VB6 from Microsoft ended during 2008. The application has been developed for over two decades and is currently in a state of developing new features and maintenance. The visual side of the application is an UI solution which is separated into page related logic regarding different views of the front-end. In figure 1 the contract overview of a test contract is shown from the local development environment.

[+ Search criteria](#)

Contract no: 3074894	Consolidate Contract:	Lessee: Virallinen nimi
Status/date: Unregistered / 21.4.2023		Reg. code: 121212-121D
Type: Instalment / End-annuity		Representer: ---
Other contracts		client ID: 3331252

Establish | Financing | Assets / Collaterals | Schedule | **Contract overview** | Memo/Role/History | Invoices | Update | CB Reporting | Vendor compensation | Activate

Contract overview | Remittance | Paid remittance

Relationship manager

Vendor contract	3074746		
Buyer profile name	EEBUEURIB3MOFINAL0		
Application / Booker	55000001 / Web Service Proposals		
Insurance	No valid insurance		
Date of entry	20.4.2023	Entry month	04.2023
Signing date		Number of instalments	120 months left / 120 m original
Financing date	21.4.2023	Payment period	6 m / 3 m
Activation date		Payment day	1
Last due date	1.7.2011		
Sub status		Collection block	No Collection block
Termination date		Dunning fee	Dunning fee will be collected

[+ Balances](#)
[+ Interest calculation](#)
[+ Maturity](#)
[+ Basic information of financing](#)
[+ Advance letters](#)
[+ Other information](#)
[+ Information on limit commission calculation](#)
[+ Audit trail-sums and cumulative balances](#)

[Cancel activation](#) [Copy contract](#)

Figure 1. Contract overview view taken from one of the test contracts generated used by the tests.

The visual elements have been created using ASP.NET Web pages [3] together with custom built components that validate user inputs. Information between the database and the front-end often flows from the ASP.NET front-end elements to SQL stored procedure calls which return the required information back, is validated by custom components and then shown to the user. A separate batch processing solution handles the logic regarding the back end of the solution, and components from .NET projects containing utility functions can be called from the front-end when more complex calculations are required that can't simply be queried from the database or calculated by the front-end solution.

Microsoft SQL Server [24] is used as a database management system and Microsoft's SQL Server Management Studio (SSMS) [9] is used to manage the SQL infrastructure. SSMS is used for both querying and managing the databases for testing purposes and creating stored procedures, functions or other SQL tools which can later be called from other parts of the application to manage data. Microsoft SQL server also ties to the SSRS technology that is used, which is a predecessor to the modern Power BI currently widely used in business intelligence related work. SSRS (SQL Server Reporting Services) [30] is used for creating and managing paginated reports based on the stored data. The reports are created by working with a Report Definition Language (RDL) file, that determines how a report visually looks and what information is displayed. The RDL file uses stored procedures as data sources to get the data that is required for the reports. SSRS reports can be generated in many different types, such as web pages, downloadable pdf- or Excel-files These reports are generated automatically daily and monthly within a batch processing side of the application but can also be generated on demand by the users from the application if required.

Git [13] is used for version control, but for most of the development time Subversion [2], also known as SVN, was used for version management. Git repositories are hosted in Azure Repos which is a part of the Azure DevOps services that are used within the project.

4.2 Application context

The application is owned by a third party and is being developed and maintained by consultants from Evitec. The application is used by financing professionals and handles information regarding different financing related information such as contracts and clients related to the contracts. A contract contains information such as what are the parties involved

within a contract, what is the contracts type, what financial assets are involved within a contract, what is the state of the contract and how terms have been negotiated between different parties involved. Contracts change over time by user actions and naturally as time goes by.

The application is split into three separate versions that all work close to one another but contain small differences on how the rules of a contract work and what features or options the users have when managing information within the system. The codebase is not however divided into separate versions. When a change is being made to one version of the application, the changes are also included in other versions as well. Functionality targeting a specific version is implemented by programming the version specific features to require the user to be using the specific version of the application. If version specific logic is developed for a contract view, the contracts in another application versions are not affected. This way there is no need to maintain three separate code bases for a very similar type of application with only minor differences. This does introduce complexity to programming and managing new features however, since functionality can be nested to conditional blocks that separate the functionality based on the application version.

Features developed are often related to working with the user interface (UI) or the reporting section of the application. Users of the application may require the reports to contain new data or different data filtering, and the changes are often application version specific. These changes require working with the RDL files and the stored procedures that query the data for the report. New UI features may require the developer to change a view in the application to fit the needs of the users better by adding additional search parameters or additional fields of information in the views.

5 Test framework

This chapter introduces the test project that is created to be later ran automatically within the Azure Pipelines system. The chapter introduces the technology used to build the test project, discusses challenges that were faced and solutions to said challenges. Architecture of the test framework is discussed as well as the individual test cases that were built.

5.1 Test project with Selenium

Selenium [22] was chosen as the UI testing framework. Selenium offers support for multiple programming languages and is considered a mature framework that has been used for over 20 years in browser automation related development. Initially the possibility of doing the tests with Visual Basic was a factor when choosing the framework, although C# was used in the end. With Selenium, UI tests can be run on multiple different browsers. Running Selenium tests requires the Selenium library to have a browser specific driver present within the project. Drivers for browsers such as Microsoft Edge, Google Chrome, Mozilla Firefox, or Internet Explorer can be imported to the testing project, which allows the creation and using of the driver object in the test project. The drivers can open the browser and control it by the actions specified by the developer in the test project. Initially, Microsoft Edge was considered as the browser used for running the tests but at the time it did not support running the tests in a headless mode. In headless mode the browser does not open a graphical interface which is particularly useful when testing web pages or web scraping. Headless Chrome has been a feature in Google Chrome since the Chrome 59 version which was released 2017 [12]. Chrome was chosen for the browser to run the tests on [12]. Later during working with the automation part of the project, it was apparent, that since the tests were ran on an agent that was installed in the local test server, choosing chrome with the headless functionality was the correct choice.

The goal of the test project is to catch software faults and software errors that new builds of the application might cause. Software faults are cases where the application does not function as expected by the requirements. Software errors occur if an action results in a situation where the application can't recover, and a task can't be finished.

A new Unit Test Project was created with Visual Studio 2022, which is a skeleton project containing MSTest unit tests. At the beginning the Selenium library was installed as well as the required browser drivers. Initially only a few Selenium test cases were created to ensure Selenium can be used to test the application. A simple login test was done with the chrome driver successfully by interacting with web elements based on their unique identifiers. A few simple test cases were created for opening a contract and browsing the application views by using hard coded contract numbers and only one specific version of the application. For verifying the test being successful, a web element was confirmed to have correct type of text after opening the contract. The test would fail if the contract was not opened correctly since there would not be a web element to show at all, or if something in the process led to a situation where the web elements could not be interacted with.

At this point it was noticed that some of the web elements that were interacted with did not have valid id values or other clear identifiers that the web driver could refer to. In these cases, the XPath of the web element was used. In some cases, interactions with the web elements were unsuccessful. The references to the web elements were seemingly correct, but interaction between the driver and the web element was unsuccessful when running the test. Even with the first few tests, workarounds were required when trying to interact with a simple button or form element. One reason for this might be that the ASP.NET generated web elements that can be difficult for Selenium to interact with.

5.1.1 Page Object Model pattern

When a software project grows and complexity increases, architecture of the project becomes increasingly more important to keep the project maintainable. The same is true also for test frameworks. In “Selenium WebDriver 3 practical guide: end-to-end automation testing for web and mobile browsers with selenium WebDriver” [14] Gundecha notes that when an automation framework is designed it needs to be accepted that it is inevitable elements within a webpage change over time. For the maintainability of the tests it is crucial that the refactoring required is minimal when these inevitable changes occur [14]. To keep the growing test project well organized a system is required to keep the project maintainable as more cases are introduced. The test project uses the Page Object Model (POM) design pattern. The idea behind POM pattern is to divide the project into “pages” and “tests” classes.

A page class contains web elements and methods that are part of a specific page within the application. Web elements such as forms and buttons that are part of a view are gathered under the same page class. Web elements are interacted with the Selenium Webdriver by using locators that point to the web element's id, XPath-address, tag, name, or other attribute that identifies the element. Actions that users do in the page can be turned into methods that do page specific actions that can be later reused across different tests. For a browsing page which the users use to search contracts or clients it can be useful to create a method for searching a client that takes the contract number or client number as a parameter. This method can then be reused by the tests that require the use of the browsing page for its contract or client searching functionality.

Test classes contain the actual test cases and test specific configuration related methods. A test file can contain for example all contract browsing related tests which often use the same page classes. The test class contains `TestInitialize`, and `TestCleanup` methods, which contain actions done before a test method is ran (`TestInitialize`) and actions done after a test method has been ran (`TestCleanup`). `TestInitialize` functions in the project contain setting up the Chrome Webdriver and initializing all the page objects that the tests would require. `TestCleanup` contains actions done after the test method was executed. `TestCleanup` in the project was used to close the drivers that were set up at the `TestInitialize` function and set the generated contracts used by the tests to a state where they would get later deleted by the batch processing.

Five separate test classes were created for the test project and a total of 20 different page classes used by the test classes. Methods were created for the page classes when creating the tests themselves and noticing a set of actions on a page that would get used by tests multiple times by moving the code from test class to the page class and creating a new method with an explanatory name. Figure 2 illustrates how the division to pages and tests is done by using the POM design pattern.

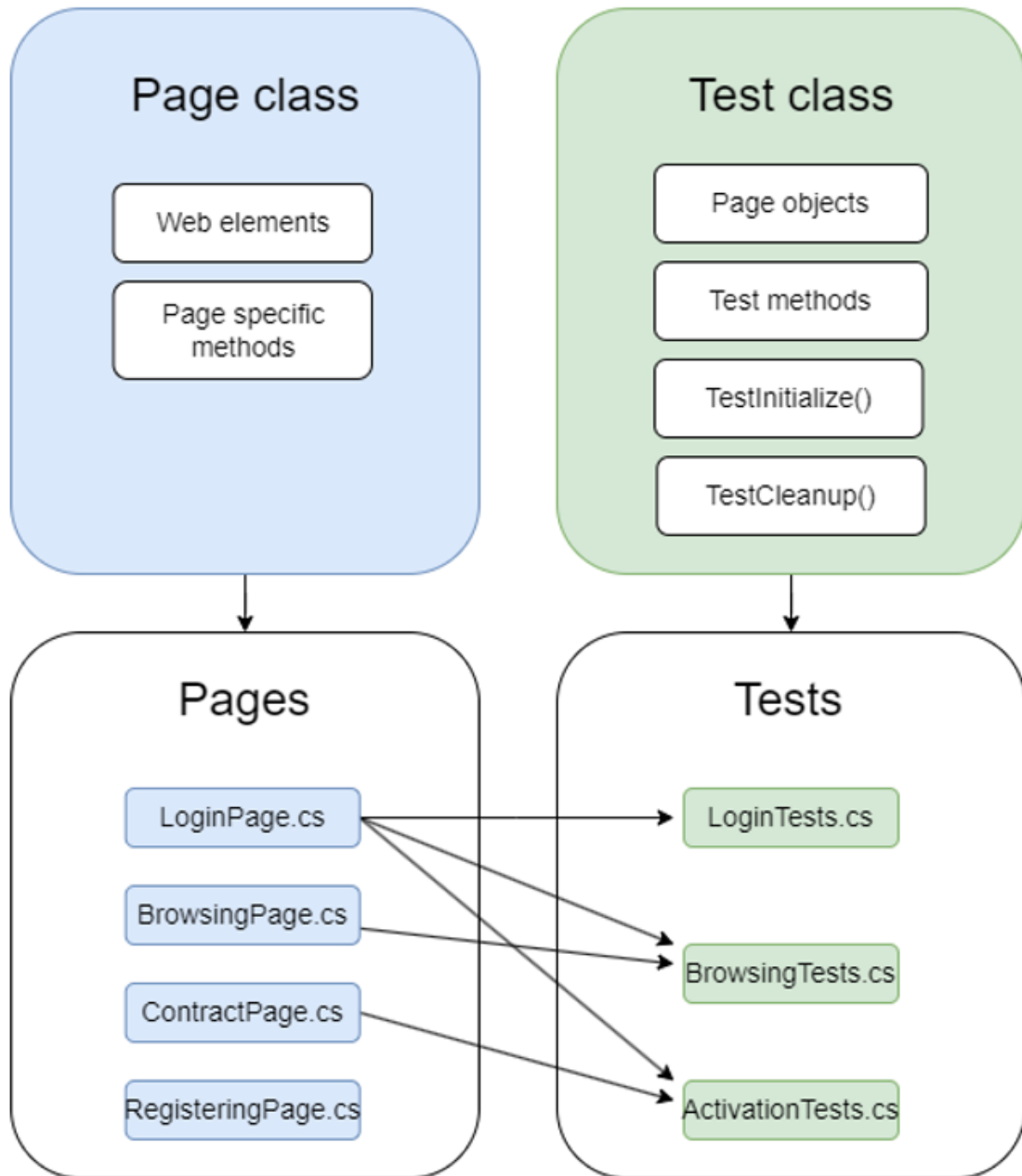


Figure 2. Page Object Model pattern used in the test project visualized. The figure contains only a couple page and test classes.

5.2 Test cases

The test cases that were created focus on browsing and contract related actions that are often used by the users. 18 different test cases were developed for the test project. Table 1 contains

all the separate test classes that contain multiple test cases, and a description of what functionality is being tested.

Table 1. Test types and description of what functionality the test case is targeting.

Test type	Number of test cases	Description
Logging in to the application	3	Three different test cases related to logging in to all three different versions of the application. Logging in to the application is also the first step for most of the other tests.
Contract browsing related tests	4	Opening the contract and testing specific actions with the opened contract. Opening all the tabs and views a contract has and testing actions that can be done to a contract.
Contract state related tests	3	Three separate tests for first generating an unregistered contract, establishing (registering) an unregistered contract, and finally activating a registered contract. Activating and establishing a contract requires actions done for the contract to initialize it for these actions. These actions are also done in the tests between generating, establishing, and activating a contract.
Activating multiple unregistered contracts simultaneously	3	Working with multiple different contracts simultaneously requires a set of rules between the chosen contracts to apply and is done in a specific consolidated contract state. This is among the newer features introduced to the application and leads to a set of contracts being grouped together.
Generating a specific report	5	Generating five different reports by navigating to the reports section of the application. The reports were created on very basic input information. The success is verified by a message from a web element that states

		that the report was started successfully. These tests do not consider the contents of the reports, only that the starting of the reports works from the application.
--	--	--

The test suite would take approximately eight minutes to run locally against the development environment, but some variation between runs was noticed when running the test suite. Logging in to the application sometimes took longer than normally, and specific contract views would take more time to open. Variation could be caused by other developers also running their own development related tests on the environment which could slow down the application and affect the tests.

5.3 Challenges in testing

Soon after the test design was started, it came apparent that there was going to be challenges regarding running the same tests multiple times. As mentioned before, contracts live within the system and their state changes over time. This affects the succession of the tests and would require the developers to change the tests that are being used or find other ways to keep the test specific contracts viable in the future. Without working with the contracts, test cases could only be made for browsing other, not so relevant application views, but since the actions with the contract are essential for the users, solution for the problem was required.

Solving the problem would require generating new contract data every time a new test run is started. This was found out to be a complex problem since creating a contract with all the required client, invoice and other contract related data would need to be generated as well and then cleaned afterwards. Creating a new contract in an “unestablished” state by running custom made SQL commands at the start of the test was one possibility but was considered too complicated solution for the purpose. This would require adding test data to many different SQL tables and after the tests have been ran, new information to related tables would have been added which would require tracing all this information and deleting it after the test run. Another possibility would have been to create separate databases only for the tests, but since the focus was on automating a test framework to the development process and not creating the best possible test framework with ideal test data management, it was not pursued.

5.3.1 Test data generation

Generating the test data was solved by using an existing service built for the application in the past. The contracts in the production version of the application get ordered from a separate third-party service. This creates a new contract base in an unestablished state that can later be registered and activated by working with the contract using the application. The contract first needs to be selected by a user and registered before the contract starts properly living in the system. Since the same third-party service used in production can't be used by the developers to order contracts for the development versions of the application, a separate existing service has been built where a new contract with the required contract and client related data is created for the test environment by simulating the service used in production. The service was tested from the test project by adding it to the project as a service. This approach solved the problem of the tests requiring a viable contract.

Ordering a new contract was set up to the `TestInitialize` method for test classes containing test cases that require working with a contract. The tests use the generated contract to run the end-to-end tests and after the test has been ran, regardless of the passing or failing of the tests the generated contracts would be set to a state where the batch processing side of the application would later clean data related to the test contracts. A stored procedure was found from the SSMS which contained thorough deletion logic of contracts, clients, and other objects within the system. This was found out to be ran every day automatically by the batch processing, and it would delete cancelled or dated contracts that were no longer relevant keep in the system. Data related to a contract gets deleted systematically from all the tables that contain data that was created by generating a contract and working with it on the application.

Using the stored procedure was found out to be good solution for the test projects data problem, since by only changing a specific date of the generated test contract, everything related the test contract would get thoroughly cleaned up without having to create new separate deletion logic. Figure 3 visualizes how a test contract is first generated in the initialization of a test, used by the test, and finally cleaned up by the `TestCleanup` method.

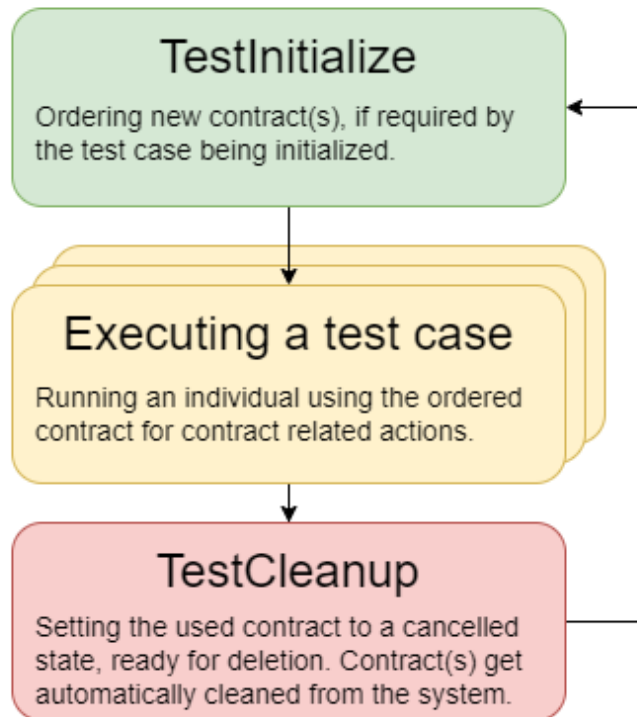


Figure 3. Visualization of the test contract's life cycle within the test project.

6 Configuring the CI/CD pipeline in Azure DevOps environment

Tests were automated by using different services within the Azure DevOps platform. Azure DevOps offers tools for software developers and companies for project management, source control, version control, development pipelines and test planning. In this chapter the role of Azure DevOps is defined within the applications development process automation and more specifically within automating the tests. [4]

6.1 Azure DevOps workflow

The application is being developed by using Azure DevOps services. Azure DevOps has a role in the development process from the moment the new task is defined by the client owning the application to the point where the feature or changes has been delivered and installed to the production environment. The following is an example of the workflow within the Azure DevOps environment of a developer creating a new feature or making changes to the application.

1. Azure Boards task is created based on the requirements gathered from the client. The task contains requirement information and technical instructions aimed towards the developer that start to work on the task. If the task contains multiple different parts, the task can be divided into separate child tasks for clarity.
2. A developer is assigned for a task by assigning them to the board task. The developer is also notified of this by the system.
3. The developer examines the boards task and creates a new branch that is based on the latest version of the master branch.
4. The developer makes the required changes to the source code and commits these changes to the task specific branch.
5. The developer tests their changes and opens a pull request from their task specific branch to the main branch.

6. The changes in the source code are reviewed by another developer before the pull request is completed.
7. Pull request is completed and the main branch now contains the new changes.
8. Boards task is completed, and a short description of the changes is documented directly to the description section of the Azure Boards task.

After these steps the latest version of the source code can be pulled from the main branch and a new build of the application can be created manually which can then be deployed to the development environment and later to the staging and production environment. At this point the changes in the source code have only been manually tested by the developer that completed the task and possibly by another developer that reviewed and tested the pull request. Creating a new build of the application and deploying it to the development environment is a process that takes time from the developers and the steps within the process are usually the same. The application has been brought to the Azure DevOps environment only recently and all the features have not been fully utilized in the project. Tools within the Azure Pipelines could be used for automating the development process, but to also add automated tests for ensuring better build quality.

6.2 Azure Pipelines

The Azure Pipelines service is used to automate the development process of the application. The automating parts consist of two separate services from the Azure DevOps Pipelines features, Pipelines and Releases. The build pipeline feature within Azure DevOps system is referred as the “Pipeline”, while the release pipeline section is called “Releases”. For simplicity these separate services are from now on referred as “build pipeline” (Pipelines) and “release pipeline” (Releases). Build pipeline handles the everything from the moment the pull request is manually completed to the point where a packaged build of the application has been created and published to Azure DevOps environment. Release pipeline handles actions regarding the deployment of the application by using the Azure Artifact created by the build pipeline.

Actions within the build pipeline are managed by editing an YAML configuration file directly from the Azure DevOps environment. The release pipeline is controlled by a user

interface which can be also turned into YAML commands if required. The entity that simulates the user that is running the pipeline actions automatically is a software component called an Azure Agent [5]. The agent can be self-hosted by the organization using the Pipelines features, or alternatively Microsoft-hosted agents can be used. While the self-hosted agents can be configured as required for the purpose, Microsoft-hosted agents also come in many different pre-configured sets of tools installed. During the project the Microsoft-hosted agents were initially used, but during the development the team switched to a self-hosted agent that runs on the local development server and was solely reserved for the teams' purposes.

6.2.1 Build pipeline

The build pipeline builds the application based on source code of a specific branch. Tasks that the development team carries out are developed on a separate branch and when changes are merged to the main branch by a pull request, the build pipeline is automatically triggered. The build pipeline can also be manually triggered by a developer, which is useful if a specific build for a specific environment is required. Usually, a separate git branch is created if an installation package is prepared for the staging or production environment, and the newest changes that are currently being tested in the development environment should be left out, or if only specific commits need to be cherry picked for a package.

The logic for the build pipeline is defined in a YAML-language which contains commands for compiling the application and managing the binary data that was produced by previous pipeline actions. The pipeline contains some conditional checks regarding the parameters that are used for building a specific version of the application. By running the pipeline manually, a developer can choose if the build created targets the development, staging or production environment and what parts of the application need to be compiled. If changes are only user interface related, there is no need to also compile the batch solutions and the reports. Figure 4 presents the options menu when manually running the build pipeline from Azure DevOps.

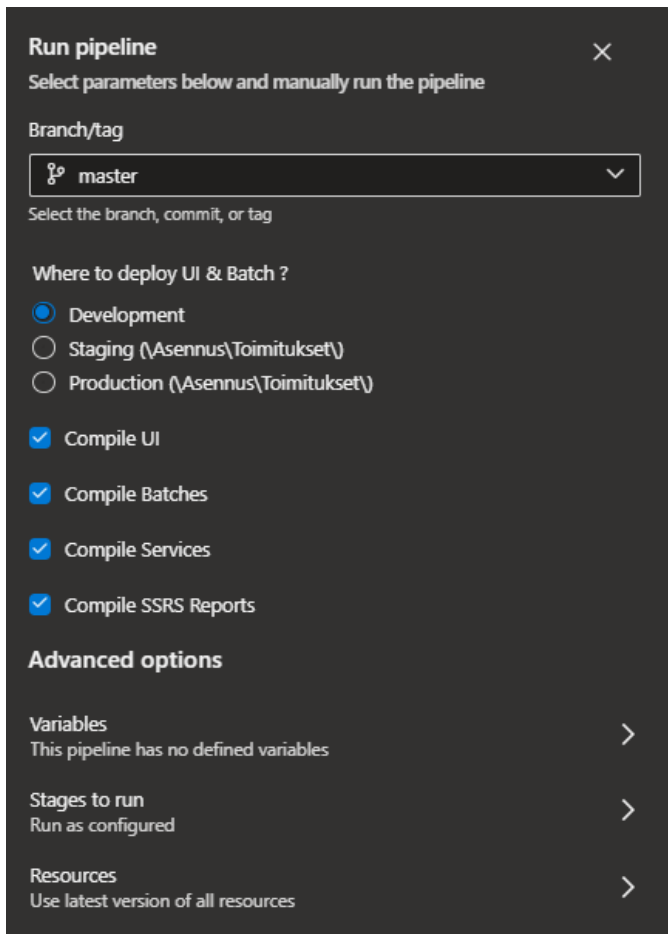


Figure 4. Options available when running the build pipeline manually.

Actions within the build pipeline are managed by editing an YAML configuration file directly from the Azure DevOps environment. The build pipeline starts by downloading the latest source code from the repository and by downloading necessary packages. Chosen projects are then compiled and tests to the database are ran to ensure database integrity. The binaries are then compressed and published as an Azure Artifact to the Azure DevOps environment. The artifact is simply a package that contains the binary project files that can be used to deploy the application. Figure 5 presents the flow of the build pipeline from the pipeline triggering automatically or manually by a user to the publishment of the artifact.

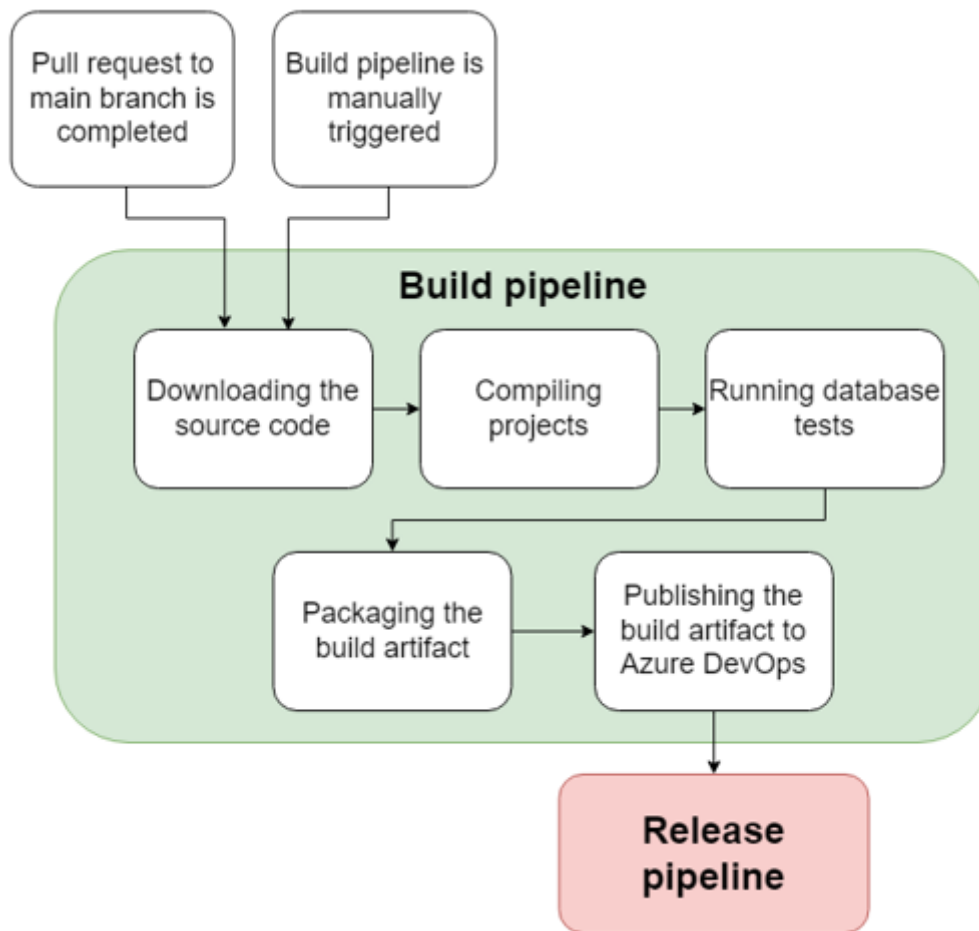


Figure 5. Build pipeline visualized

Among the other projects that are compiled within the build pipeline the new test project was also set to be compiled. The test project lives in the Azure Repos version control with the rest of the application but is only compiled and included in the packages on builds that target the local development environment. This was achieved by adding conditional checks to the YAML configuration files controlling the pipeline actions. This was a necessary constraint to for the build pipeline, since the tests are not going to be ran on the staging environment or the production environment, so adding them to the deliveries for other environments would be both unnecessary and possibly dangerous. Staging and production environment targeted builds can only be created by the pipeline if a developer manually selects one of these options. If a developer approves a pull request to the master, the pipeline always builds and publishes the application to the local test server. The following figure 6

contains the list of all the separate YAML based build pipeline actions that have been created by the development team.

Jobs		
✓	Compile & build packages	3m 30s
✓	Initialize job	2s
✓	Checkout @master	19s
✓	Restore nuget UI	12s
✓	Building UI	22s
✓	Building batches	22s
✓	Building Services	3s
✓	Building reports	9s
✓	Restore nuget Tests	5s
✓	Building Tests	6s
✓	Test database integrity (create e...	27s
✓	Update databases using Roundh...	11s
✓	Copy installation script	1s
⏸	Copy installation instruction guide	<1s
✓	Create additional installation fold...	1s
✓	ZIP UI	26s
✓	PublishBuildArtifacts	6s
✓	Post-job: Checkout	<1s
✓	Finalize Job	<1s
✓	Report build status	<1s

Phase	Tasks
Initialization	Initialize job, Checkout @master
Compiling projects	Restore nuget UI, Building UI, Building batches, Building Services, Building reports, Restore nuget Tests, Building Tests
Database tests and updates	Test database integrity (create e...), Update databases using Roundh...
Package management and publishing the Azure Artifact	Copy installation script, Copy installation instruction guide, Create additional installation fold..., ZIP UI, PublishBuildArtifacts
Finalization	Post-job: Checkout, Finalize Job, Report build status

Figure 6. Individual build pipeline tasks in order of execution.

Successful run of the build pipeline results in the pipeline publishing an Azure Artifact to the Azure DevOps environment. The artifact is basically a collection of binary files bundled together that can be used for installing and deploying the application later.

6.2.2 Release pipeline

The release pipeline continues the pipeline tasks from where the build pipeline left off by doing tasks with the Azure Artifact that was published. The release pipeline was configured to be automatically triggered after the build pipeline finishes successfully. The starting point for the release pipeline is the previously created artifact, which is first downloaded. The release pipeline consists of two separate stages that are run one after another. Stages are groups of actions that contain tasks that an Azure Agent can run. The release pipeline view containing the stages is presented in figure 7.

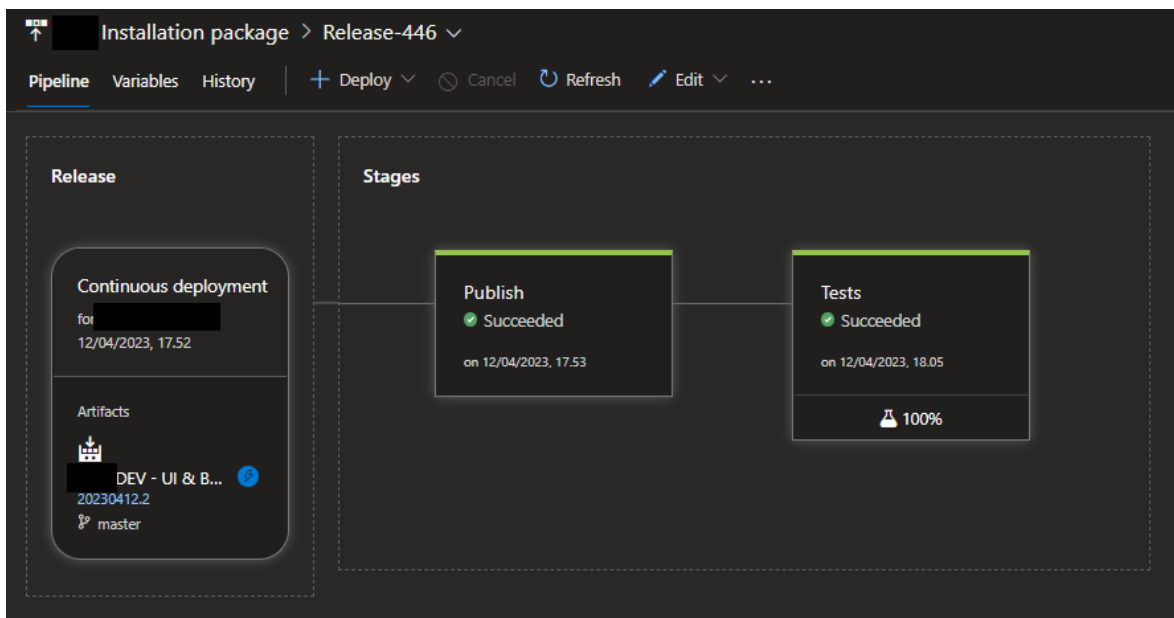


Figure 7. Stages of the release pipeline. The “Release” section shows the artifact that triggered the release pipeline to run the two separate stages.

The release pipeline is divided into two separate stages “Publish” and “Tests”. Publish stage deploys the application to the local test server by managing the binary files and installing the application. If the publish stage is completed successfully, the Tests stage is triggered afterwards, which runs automated end-to-end tests on the published application. Tests stage first installs a Visual Studio Test Platform that allows the Azure Agent running the tests on the development environment to run visual studio testing commands without having the

complete Visual Studio packages installed on the local test server and tries to find tests from the artifact according to a given path and if found, runs these tests one after another. The test results can be viewed live during the test runs from the release pipeline, or reviewed from the Azure Test Plans section of the Azure DevOps environment after a test run has been completed. The flow of the release pipeline is presented in figure 8.

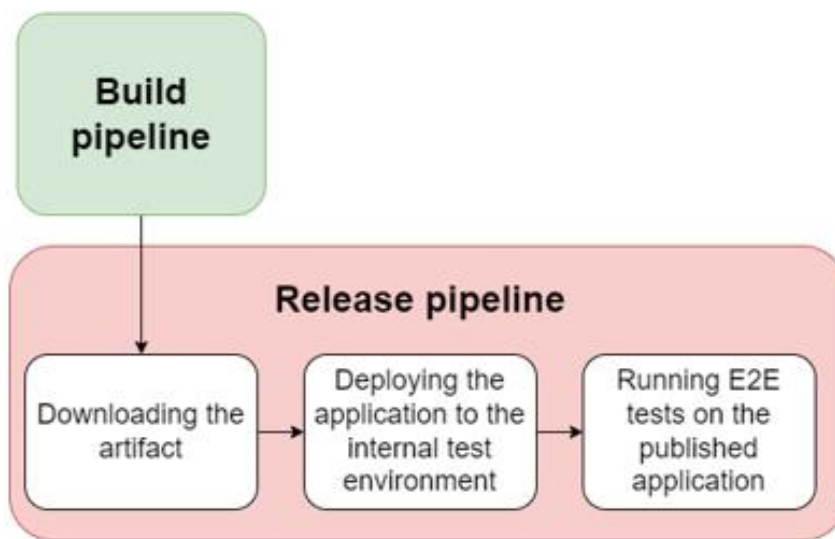
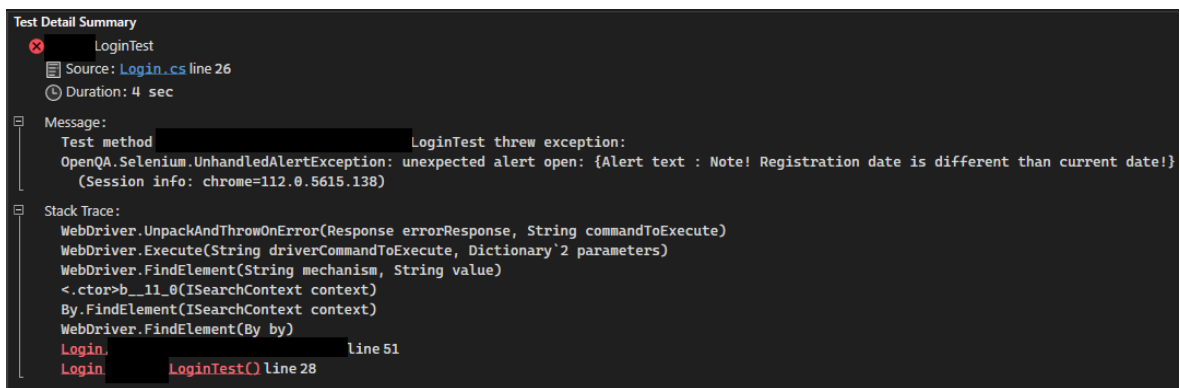


Figure 8. Release pipeline visualized

6.3 Release pipeline test results

Testing results are gathered under the Test Plans section of Azure DevOps after the release pipeline has run. Test Plans offer a lot of functionality and tools for designing and documenting a test framework, attaching the tests to their respectable work items or user stories, and visualizing the test results of the project with informational charts. Runs section within the Test Plans contains information about the test runs, and individual test case logs can be reviewed. Runs also contains a downloadable TRX-file which contains test result information and can be downloaded and opened with Visual Studio if more detailed information is required while analysing the results.

Varying results were found when analysing the test results. With some cases the reason for test failure can be easily understood from the test detail summary as in the case of figure 9, where an alert window blocked the Selenium Webdriver resulting in an exception being thrown. The specific code line where the test fails can be read from the last row of the summary, but the message section offers enough information to know that the registration date within a certain application version is set incorrectly, which can be caused if end-of-day or end-of-month batch processing routines fail.



```

Test Detail Summary
LoginTest
Source: Login.cs line 26
Duration: 4 sec
Message:
Test method [redacted] LoginTest threw exception:
OpenQA.Selenium.UnhandledAlertException: unexpected alert open: {Alert text : Note! Registration date is different than current date!}
(Session info: chrome=112.0.5615.138)
Stack Trace:
WebDriver.UnpackAndThrowOnError(Response errorResponse, String commandToExecute)
WebDriver.Execute(String driverCommandToExecute, Dictionary'2 parameters)
WebDriver.FindElement(String mechanism, String value)
<.ctor>b__11_0(ISearchContext context)
By.FindElement(ISearchContext context)
WebDriver.FindElement(By by)
Login [redacted] line 51
Login LoginTest() line 28

```

Figure 9. Test results from a failed login test case

On other cases the detail summaries were often not so easy to understand. For someone who had not worked with the tests themselves, only a reference to the line of code where the test failed does not immediately tell what went wrong with specific test case. If the reviewer of the testing results has not worked with the tests themselves, understanding what caused the failure can be difficult. These problems were especially apparent on situations where the web application was not ready for the actions of the Selenium Webdriver, and the test would fail.

7 Results and future development

A test suite was created successfully by creating a Visual Studio project and using Selenium Webdriver to create end-to-end test cases that test typical tasks that a user would perform using the application. The tests were later automated successfully to the development process by using Azure DevOps Pipelines services. Testing results were gathered by leaving the pipelines to run automatically as new versions of the application were deployed by the pipeline. It was noticed that often the test results that the automated tests produced that were not successful, were often failing because something else than the application itself not functioning correctly. The fails appeared from situations where the test itself needed changes to be more stable, while the logic that was targeted by the test would function properly. The web application under testing normally runs slower for a while after a new version has been published to the development server. This interestingly did not affect the testing results in negative ways. The tests would just run for longer as the agent running the tests would have to wait longer for the web application to open different views during the first tests. This was unexpected since before agents were running the tests automatically, difficulties with the application views keeping up with the tests were a problem and required workarounds. After deployment the first time the agent ran a logging in test, it could naturally take for quite a while, over a minute even. This slowing down might be caused by the cache clearing on every deployment which slows the application initially or warm up periods that an ASP.NET Web page can have after deployment.

Users of the application gave feedback on the solution when the results were presented and multiple new ideas for testing related ideas came up related to creating browser ran tests that can act as sanity checks for new deployments to the staging environment. Other possibilities regarding integration testing were also discussed for the future. This would probably be better suited for a separate project where the focus would not be on running the tests on the browser, but because of the pipeline development that had went into the project, new tests could be possibly quite easily incorporated to the existing pipeline.

Some difficulties were faced during the development regarding getting the configuration language YAML to work together with the build commands that had been used to build and publish the application. Some of the commands used for deployment on the command

line before CI/CD development did not work automatically, but by either using newer build commands or changing parameters, these problems were eventually solved by research, trial, and error.

When working together with other team members, there was often confusion regarding the terminology in the Azure DevOps platform. The artifact being created by the build pipeline to the Azure DevOps environment was referred to as “publishing” the artifact, which is quite confusing since within other Microsoft systems publishing is used as a synonym to deployment of the application itself to a server. Sometimes when developing publishing features to the build pipeline another team member would think that this is development regarding the deployment of the application and not about publishing an Azure Artifact to the DevOps environment. The role of the Azure Pipelines and Azure Releases regarding the process was also unclear, and the help found in the web had often conflicting information regarding which pipeline features to use for what purpose. After the practical development had ended, it was understood that the Releases section of the Azure Pipelines features is something that Microsoft might want the users to steer away from, and the initial YAML pipeline could and probably should be used for the entire process.

The resulting release pipeline was quite simple that only contains the bare minimum for getting the application published to the local development server and then running the tests automatically. In future development releases side of the Azure DevOps could be used to represent the lifecycle of a version of the application from it being introduced to the local development environment to eventually being released to production. During development to run the pipelines, Microsoft hosted Azure Agent was used for testing, but during development this was changed to a locally hosted agent. Only one local Azure agent was used for both the build pipeline and the release pipeline. This led to a situation where sometimes the agent would be occupied to run the Selenium tests and the build pipeline would have to wait for the agent to be free again. Adding more Microsoft-hosted agents in the future or using a different agent for running the automated tests and the building and packaging part of the pipelines could solve this problem.

7.1 Limitations and validity

Research presented has some limitations regarding the validity of the results. Although some different tools for testing were considered and researched before starting to develop the test framework, no comprehensive study for all the industry standard testing frameworks was conducted. Reason for this was time and resources that were allocated for the study as well as the scope of the project in the first place. Automation and development operations was focused more during the development, and tools used for developing the CI/CD pipeline were determined by what was currently being used within the project. Quality of the test framework produced, and the development process automation introduced within the project are also affected by the fact that most of the topics that were worked on by the developers were new for them and there was no clear understanding of the best practices regarding building a test automation framework or CI/CD pipeline automation, only ideas of what could be done with the tools and what would benefit the current development and testing processes.

8 Discussion

The software project did gain improvements regarding the automating parts of the development. Before the development with the Azure Pipelines, new builds were published to the local test environment by manually compiling the source code with environment specific settings and then publishing the build to the environment. This took a lot of time, and after the development the team always knew that if the pipeline did not fail, the development environment always had the most recent version of the application. The most value gained seemed to come from working on the Azure DevOps Pipelines systems. To get the tests automated properly, the pipeline required a lot of work and testing with compiling the application and then publishing it to the test environment automatically before the test project was included. Lai et al. listed [18] five different phases solid CI/CD pipeline may consist of. The resulting pipeline incorporates three of these phases listed: build phase, deployment phase and automated test phase. The project does not have a separate unit testing solution, so creating automation for unit tests is not relevant for the project. Pipeline automation is also constrained by the fact that the product is deployed to an environment controlled by a separate third-party which is not directly accessible to the consultants working on the source code.

Clever usage and further development of the Release section of Azure DevOps Pipelines could help manage the different builds and deployments that live in the development, staging and production environments. Releases allows the creation of a model consisting of multiple stages and can even contain manual confirmations by team members between stages. With the tool the entire flow of the application versions moving from one environment to another could be visualized and this could considerably help tracking the deployments. Automating the deployments for all the three environments has its problems since the staging environment and the production environment deployments are not directly controlled by the team that actively develops the application. Releases could however be used to monitor the lifecycle of a new deployment package containing changes to the application. This allows the developers to track what features have reached which environments and which of the stakeholders within the project have tested what features. The following figure 10 describes how the release pipeline could be created within the system.

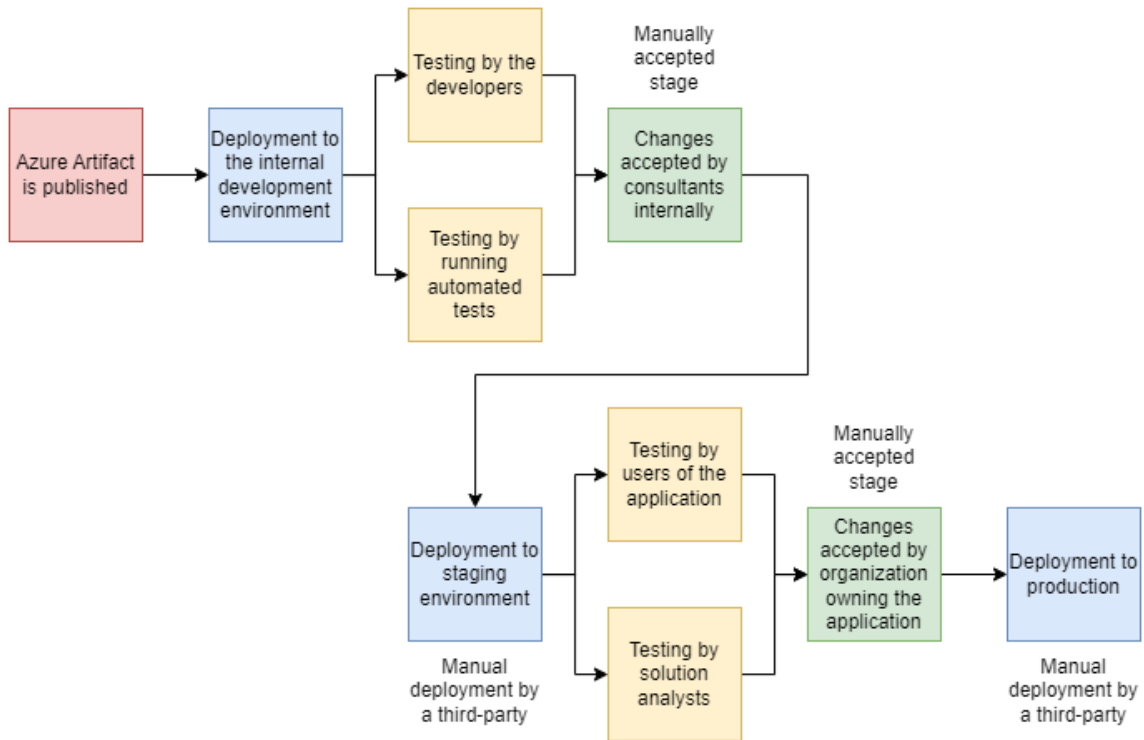


Figure 10. Visualization of how a further developed release pipeline could work for tracking the deployments

Automating the deployment process does however collide with a problem that the application changes currently do not always follow a clear protocol as presented in the figure 12. Some changes can leap over the staging environment directly to production and sometimes testing in the local development server can be minimal if the changes are very straight forward and require to be deployed to production fast. The requirements from the client might require fast actions and sometimes changes need to be done directly to production. This could however help keeping track of regular tasks that are not hotfixes directly to production but worked on for longer periods of time. A new feature might consist of many separate stages and require development from multiple developers in charge of their own parts of the work task. Problems regarding the continuous deployment within the project were both technical and social. Similar challenges were discovered as found in [7] where technical as well as social challenges affect the continuous deployment part of the DevOps process. Getting the whole team on the same page regarding the practices within the CI/CD process can be difficult even within smaller development teams.

An important aspect that was not considered during the development came from feedback of other team members when presenting the tests and test results. As test result section of the Azure DevOps was presented, a team member brought up a question regarding how a developer who had not worked on the test project itself would understand what went wrong with the application if a test fails. Solution to this could come from involving other team members to the test project more and eventually work on the tests themselves. Another idea that was discussed was about taking screenshots of the application at the time of a test failing. The screenshot could be referred from the test results for better understanding exactly at what point the application failed to pinpoint the problem faster. Prototype for this feature was tested with the login tests. After the Selenium WebDriver had successfully logged in to the application a screenshot would be taken from the browsing view that is opened by default after logging in. This worked surprisingly well, even if running the tests in the headless mode by the Azure Agent. In future development this feature could be used for taking a screenshot automatically when a test fails just before closing the browser after failure.

This pointed out the difficulty of introducing testing late in the software development lifecycle during the maintenance phase. The team would need to introduce producing and maintaining the tests while developing features themselves, although currently development and testing were seen as clearly separate activities. Development as something that creates new features and testing as the activity that confirms the developed features function correctly after deployment. Cruzes et al. pointed out the importance of manual tests and the risks of automating too much activities within the DevOps process and discarding manual testing altogether [8]. Based on the projects current level of automation, more focus should probably be given to deployment and testing task automation without trying to completely get rid of manual testing practices.

Flakiness within UI tests effected the test results considerably. As other research suggested [19, 21] this is a typical problem when working with end-to-end testing. Flakiness led to false positives and false negatives on results, which decreased the reliability of the test results. Browser tests were not found out to be the most useful for finding faults from the application functioning incorrectly but would notify of other problems regarding the environment where the application runs or the tests themselves requiring changes to increase their stability. If testing efforts were to be continued, different types of tests that target the most vulnerable parts of the application that can most likely be affected by the current

development should be focused on. The fact that the application is split into three different versions leads to situations where a change is required for a specific version, but the new functionality should not affect two other versions in any ways. This development is very error prone and vulnerable aspects of the application that would benefit from automatized tests. The development done for the project now allows the development team to automate other types of tests in the future as well.

Automating tests was initially thought as the proper way for catching errors from the application without having to manually go through the most common routines after deployments. More value could however be gained from developing the system around the CI/CD systems in Azure DevOps and creating a more robust system for managing the versions of the application. Problems in the staging environment and in production often come from the manual work that is required for a build to be installed to a new environment. When a user interface related change that also has SQL related development is deployed to the staging environment, these changes are both separately deployed by hand to these environments. There can be a situation for example where the staging environment can have the UI related ASP.NET development already deployed but is missing the SQL changes that the front-end changes require to function which can lead to crashes or a view showing information incorrectly. These problems are impossible for end-to-end tests to catch since the errors happen in the deployment process and not in the development of the features. Focusing on the technical and social challenges that continuous delivery brings as discussed in “On the journey to continuous deployment: Technical and social challenges along the way” [7].

9 Conclusions

Test automation was introduced to an existing software project in maintenance successfully. A prototype test project was built using the .NET framework together with Selenium Webdriver for automatized browser testing. The test suite created follows the Page Object Model pattern for managing the growing project. The test project was set to run automatically within the development process when changes were made to the application. Services from Pipelines were used for configuring the tests to run automatically on the Azure DevOps environment and Test Results for reviewing and analysing the test log information produced. Pipelines was used for creating a separate build-pipeline in charge of compiling and packaging a build, while Releases was used for deploying the specific build and running the browser-based tests after deployment.

The research questions that were set to be answered in the thesis considered whether it is possible to create automatized end-to-end tests and automate the development process of an application in maintenance phase and if this offers value to the project. Creating the test project and getting it working was done successfully, but most of the value gained did not come from test results pointing out application faults, but from working on automating the development and deployment processes. A lot of the manual work that was previously done by the developers regarding version control and deploying the application internally was squashed by the Pipelines development.

Data from the end-to-end tests were analysed after letting the tests run for a while within the pipelines. The tests were found out to produce occasional false negatives and false positives. A failing test would more often indicate that something regarding the batch processing or the state of the application on the development environment was blocking the tests to run successfully. This would point to a problem that needs to be fixed but does not directly contribute to verifying that the feature that the end-to-end test targets functions as expected. Test related findings were in line with the findings from other related research that was studied before the development. Clients in charge of the applications development saw potential in future development regarding the automatized tests and future development regarding testing was seen as a possible future development.

References

- [1] Ammann, P. and Offutt, J. 2016. *Introduction to software testing*. Cambridge University Press.
- [2] Apache Subversion: <https://subversion.apache.org/>. Accessed: 2023-06-12.
- [3] ASP.NET | Open-source web framework for .NET: <https://dotnet.microsoft.com/en-us/apps/aspnet>. Accessed: 2023-06-12.
- [4] Azure DevOps Services | Microsoft Azure: <https://azure.microsoft.com/en-us/products/devops>. Accessed: 2023-01-21.
- [5] Azure Pipelines Agents - Azure Pipelines: 2023. <https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/agents>. Accessed: 2023-05-18.
- [6] Christophe, L. et al. 2014. Prevalence and Maintenance of Automated Functional Tests for Web Applications. *2014 IEEE International Conference on Software Maintenance and Evolution* (2014), 141–150.
- [7] Claps, G.G. et al. 2015. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*. 57, (2015), 21–31. DOI:<https://doi.org/10.1016/j.infsof.2014.07.009>.
- [8] Cruzes, D.S. et al. 2019. Testing in a DevOps Era: Perceptions of Testers in Norwegian Organisations. *Computational Science and Its Applications – ICCSA 2019* (Cham, 2019), 442–455.
- [9] Download SQL Server Management Studio (SSMS) - SQL Server Management Studio (SSMS) | Microsoft Learn: <https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16>. Accessed: 2023-06-12.
- [10] Ebert, C. et al. 2016. DevOps. *IEEE Software*. 33, 3 (2016), 94–100. DOI:<https://doi.org/10.1109/MS.2016.68>.
- [11] Evitec – We modernise your business with software solutions: <https://evitec.com/>. Accessed: 2023-03-27.
- [12] Getting Started with Headless Chrome: <https://developer.chrome.com/blog/headless-chrome/>. Accessed: 2023-02-24.
- [13] Git: <https://git-scm.com/>. Accessed: 2023-06-12.
- [14] Gundecha, U. 2018. *Selenium WebDriver 3 practical guide : end-to-end automation testing for web and mobile browsers with selenium WebDriver*. Packt.

- [15] Host, M. and Runeson, P. 2007. Checklists for Software Engineering Case Study Research. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)* (2007), 479–481.
- [16] JavaScript Component Testing and E2E Testing Framework | Cypress: <https://www.cypress.io/>. Accessed: 2023-06-12.
- [17] Kasurinen, J. et al. 2010. Software Test Automation in Practice: Empirical Observations. *Advances in Software Engineering*. 2010, (Feb. 2010), 1–18. DOI:<https://doi.org/10.1155/2010/620836>.
- [18] Lai, S.-T. et al. 2022. Combining Pipeline Quality with Automation to CI/CD Process for Improving Quality and Efficiency of Software Maintenance. *Innovative Mobile and Internet Services in Ubiquitous Computing* (Cham, 2022), 362–372.
- [19] Leotta, M. et al. 2023. Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey. *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), 339–350.
- [20] Memon, A.M. and Soffa, M.L. 2003. Regression Testing of GUIs. *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2003), 118–127.
- [21] Olianas, D. et al. 2021. Reducing flakiness in End-to-End test suites: an experience report. *Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, September 8–11, 2021, Proceedings* (2021), 3–17.
- [22] Selenium: <https://www.selenium.dev/>. Accessed: 2023-04-27.
- [23] Singh, Y. 2011. *Software testing*. Cambridge University Press.
- [24] SQL Server 2022 | Microsoft: <https://www.microsoft.com/en-us/sql-server/sql-server-2022>. Accessed: 2023-04-27.
- [25] The different types of testing in software: <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>. Accessed: 2023-03-10.
- [26] Tsai, W.T. et al. 2001. End-to-end integration testing design. *25th Annual International Computer Software and Applications Conference. COMPSAC 2001* (2001), 166–171.
- [27] Types of Software Testing: 2019. <https://devqa.io/types-of-testing/>. Accessed: 2023-04-18.

- [28] Vila, E. et al. 2017. Automation Testing Framework for Web Applications with Selenium WebDriver: Opportunities and Threats. *Proceedings of the International Conference on Advances in Image Processing* (New York, NY, USA, 2017), 144–150.
- [29] What is a Test Case?
<https://www.techtarget.com/searchsoftwarequality/definition/test-case>. Accessed: 2023-04-17.
- [30] What is SQL Server Reporting Services - SQL Server Reporting Services (SSRS): 2023. *<https://learn.microsoft.com/en-us/sql/reporting-services/create-deploy-and-manage-mobile-and-paginated-reports>*. Accessed: 2023-04-16.
- [31] Yin, R.K. 2009. *Case study research: Design and methods*. sage.