Bachelor's thesis

Information and Communications Technology

2019

Jukka Riihimäki

AUTOMATIC TESTING AND DELIVERY IN WEB DEVELOPMENT

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2019 | 35 pages

Jukka Riihimäki

AUTOMATIC TESTING AND DELIVERY IN WEB DEVELOPMENT

This thesis aims to investigate how web development can be automated and improved. It aims to describe topics such as development and testing processes, DevOps culture and methods of automating multiple parts of the development pipeline with workflows such as continuous integration and delivery.

In-depth research was conducted starting from development practices. Agile software development and the DevOps movement were researched, alongside with possibilities in testing and delivery. Modern web development frameworks and their technical implementations were also researched.

This thesis resulted in a proof of concept, which provides a demonstration for the points made in the theoretical sections. The proof of concept includes development workflow with continuous integration and delivery. It allows a base for developers to start building their own continuous integration and delivery project.

The frontend part of the proof of concept was implemented with TypeScript and React. The backend was created using NodeJS, Express, and PostgreSQL. It includes end-to-end testing from the PostgreSQL database to the React frontend. All technologies except the database are JavaScript-based.

KEYWORDS:

Web Development, Automatic Testing, Continuous Integration, Continuous Delivery, Continuous Deployment, JavaScript, proof of concept

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2019 | 35 sivua

Jukka Riihimäki

WEB-OHJELMISTOJEN AUTOMAATTINEN TESTAUS JA TOIMITUS

Web-pohjaiset ohjelmistot ovat monimutkaistuneet viime vuosina, sekä niiltä odotetaan paljon enemmän. Odotusten kasvaessa kehitysprosessit ja testaaminen, sekä niiden automatisointi ovat ajankohtaisia aiheita. Automatisoinnilla on mahdollista nopeuttaa uusien versioiden julkaisua, sekä tehdä kehittäjien ja ylläpitäjien työstä tehokkaampaa.

Opinnäytetyön tarkoituksena oli tutkia kuinka web-kehitystä on mahdollista automatisoida sekä kehittää. Opinnäytetyössä tutkitaan seuraavia aiheita: kehitys- ja testausprosessit, DevOps -kulttuuri sekä tavat, joilla voitaisiin automatisoida eri osia kehitysprosessista. Näistä tavoista käsiteltiin muun muassa automaattitestausta, jatkuvaa integraatiota, jatkuvaa toimitusta sekä julkaisua.

Syventävä tutkimus aloitettiin tutkimalla ohjelmistokehityksen tapoja ja prosesseja. Ketterää ohjelmistokehitystä sekä DevOps-kulttuuria tutkittiin, yhdistelemällä ne mahdollisuuksiin ohjelmistotestauksen ja toimituksen saralla. Lisäksi perehdyttiin modernin web-kehityksen viitekehyksiin sekä niiden teknisiin implementaatioihin.

Opinnäytetyön tulos on teoreettisen tutkinnan lisäksi konkreettinen soveltuvuusselvitys, joka sisältää demonstraation aiheille, joita käsitellään teoreettisissa osioissa. Selvitys sisältää valmiiksi asennetun automatisoidun kehitysprosessin, joka sisältää pohjan jatkuvalle integraatiolle sekä toimitukselle, mukaanlukien automaattitestit.

Soveltuvuusselvityksen presentaationaalinen osuus, eli selainpuoli, on tehty käyttäen TypeScriptiä ja Reactia. Palvelinpuoli on toteutettu NodeJS, Express ja PostgreSQL-yhdistelmällä. Soveltuvuusselvitys sisältää kokonaisvaltaisen testauksen PostgreSQL-tietokannasta selainpuolen Reactiin. Tietokanta poislukien, edellä mainitut teknologiat ovat kaikki JavaScript-pohjaisia teknologioita.

ASIASANAT:

Web-kehitys, Automaattitestaus, Jatkuva Integraatio, Jatkuva Toimitus, Jatkuva Julkaisu, JavaScript

CONTENTS

FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface is the definition of methods used to communicate between components. API can be used for example web-based systems and software libraries.
BDD	Behavior-driven development is a software development process that emerged from TDD. The main idea of it is intense collaboration between all team members, using conversation to formalize a shared understanding on software requirements.
CD	Continuous Delivery moves automatically new code between different stages of the software release cycle.
CI	Continuous Integration defines practice to merge new code to mainline as often as possible.
DevOps	Stands for development and information technology operations. It is a culture and a set of software development practices that aims to shorten the development life cycle while delivering pieces of software.
HTTP	Hypertext Transfer Protocol is used to transmit documents between web servers and web browsers.
TDD	Test-driven development is a software development process that relies on short development cycles. Requirements are turned into test cases which the software is expected to pass.
VCS	Version Control System helps to manage application source code and changes to it.

1 INTRODUCTION

Web services and technologies are advancing and growing in complexity. Web development is increasingly becoming like software development. Web technologies are used to create native and desktop applications with technologies such as Electron, React Native and Progressive Web Applications which are discussed in Chapter 2. Therefore, software development also requires an increased amount of attention in investing in development processes. This thesis aims to investigate modern web development and how web development processes can be improved in terms of speed, code quality and even developer experience.

This thesis aims to investigate DevOps culture, which is a movement that automates the processes between software development and IT teams so that they can build, test, and release software faster and more reliably. As a movement DevOps is becoming increasingly popular amongst IT companies and according to surveys on the issue, adapting DevOps can visibly help accelerate deployment. The time required for changes to go from a committed stage to code in production can be even 200 times faster for high performing organizations that have adapted DevOps culture. (Puppet, 2015)

To adapt DevOps or an agile mindset, it is important to look at what could be automated in multiple parts of the development pipeline. These parts can include aspects such as testing, integration and delivery. Integration and delivery are discussed in Chapter 4. Different methods of testing are discussed in Chapter 5.

This thesis aims to implement a proof of concept, which provides a concrete demonstration for the points made in the theoretical sections. The proof of concept includes development workflow with continuous integration and delivery. It allows a base for developers to start building their own project with continuous integration and delivery.

The frontend part of the proof of concept was implemented with TypeScript and React. The backend was created using NodeJS, Express and PostgreSQL. It includes end-to-end tests from the PostgreSQL database to the React frontend. All technologies except the database are JavaScript-based.

Multiple similar publications and theses around the issue can be found. A thesis focusing heavily on DevOps, “An approach to Software Deployment” (Kostecky, 2019), explores

the organizational changes required for the process. The thesis also includes a technical implementation, that was directly made for an organization.

The thesis “JavaScript Beyond the Browser” (Parshina, 2018) studies different approaches to mobile and desktop development with web technologies. The thesis also introduces a short history of JavaScript and the improvements made to the language. Multiple modern web technologies are discussed and a few of them were given practical examples.

This thesis differs from the formerly mentioned theses in a sense that it is more of a general look into development and automation processes. The project implemented in this thesis also focuses solely on web development. The project is an open source base project for anyone to use and modify. The project is released under MIT license.

2 MODERN WEB DEVELOPMENT

Adapting good testing practices and automating processes in web application development has become increasingly important due to the recent advancements in web technologies. The web is increasingly used for native-like applications with various technologies.

2.1 Web in Native

A web application reach is not only limited to web browsers. They can run for example in desktop and mobile native applications with technologies such as Electron, progressive web applications and hybrid applications.

Electron-based desktop applications are becoming increasingly popular. Electron is an open source library which combines Chromium and Node.js into a single runtime. This runtime allows web applications to run in a native desktop environment. This means that large parts of new software are completely a web based. Few popular programs that are built by using Electron are Skype, Discord and Slack. (Electron, 2019)

Progressive web applications (PWAs) use modern web APIs to create cross-platform web application. PWA is a combination of technologies and methods. For an application to be defined as a PWA it should meet certain requirements. The requirements include:

- All app URLs load while offline
- Metadata provided for Add to Home Screen
- Site works cross browser
- Pages are responsive on tablets and mobile devices
- Site is served over HTTPS

(Google Developers, 2019)

Hybrid applications are created as a single application to use on multiple platforms. Technically, hybrid applications are native applications and web applications bundled together. The program itself is a web application put in a native application container to be connected to the device hardware. (VT Netzwelt, 2018)

2.2 Separating the server and view

To achieve the use of methods and technologies such as PWAs and hybrid applications, often the need for separating the server and view arises. This can be achieved for example with decoupled design and APIs.

Decoupled design is when the backend layer of an application is separated from the frontend layer. The backend layer can be, for example, a content management system or a custom program that exposes its data for an API. The frontend can then consume the data from the API and display it to the user. These kind of frontend applications are mostly implemented with JavaScript and its libraries such as React, Angular and Vue.

Decoupled design allows for multiple frontends to coexist on the same application. This means that they consume the same data but display a separate view. JavaScript applications can also be run in different environments such as a web, desktop and mobile. This raises the need for testing components and the APIs in case their behavior changes.

2.3 Open source

> *Open source software is software with source code that anyone can inspect, modify, and enhance. "Source code" is the part of software that most computer users don't ever see; it's the code computer programmers can manipulate to change how a piece of software—a "program" or "application"—works. Programmers who have access to a computer program's source code can improve that program by adding features to it or fixing parts that don't always work correctly. (Opensource.com, 2019)*

There are multiple types of open source licenses. They are usually divided into two main categories: copyleft and permissive. The division between these two is based on the requirements and restrictions the license places on its users.

Copyleft license in short stands for a license that must stay the same, even if changes are made to its code and re-released. A popular copyleft license is GPL-3. **Permissive license** guarantees more freedom and permits proprietary derivative works. A popular permissive license is MIT. (Opensource.com, 2019)

Open source is beneficial in a sense that when a piece of software is needed, it is often already implemented and available. This increases the developer's responsibility in knowing what pieces they are using and why. It is impossible to master all tools that are used in development, but it is important to keep up with the field's standards.

As always, the rapid development of software opens a gateway for issues and bugs. Open source is a great way to counter these issues. The community has an interest in fixing these and continuous development is guaranteed. This also opens the possibility for such rapid development of software. This is, of course, when the technology in question is relevant and used.

2.4 Plug and play software

Web development has grown in a way that implementing everything from scratch takes so much time that by the time development is finished, the technologies might as well already be deprecated. Today's web development, due to the popularity of open source and its fast-paced nature, is more like building software from multiple pieces. Some examples of these pieces are discussed in the following paragraphs.

Frameworks provide a standard way to build and deploy applications to the web. They often do most of the heavy lifting in development, enabling focus on building the main parts of the application.

Component-based thinking is increasingly popular especially in the JavaScript world. Components are independent pieces coexisting in a single user interface (UI) that are developed with reusability in mind. Ideally, each component contains their own logic and API calls. Ready-to-use components can be found in many component libraries around the web.

Automated testing and continuous integration tools are means to delivering quality software faster. Multiple tools can be found in the open source community. These tools will be further discussed in later sections.

3 DEVELOPMENT WORKFLOW

The development workflow is the process of code being transferred from a developer's local environment to the common code mainline. It contains multiple moving parts of which some can be automated in various ways, speeding up the development process.

3.1 Version Control System

A version control system (VCS) is also known as a revision control or source control system. It allows users to keep track of the changes in projects, enabling multiple users to collaborate in a more reliable manner. Tasks can be separated through branches, keeping one branch as the mainline branch.

Change history can be useful in multiple cases. A good change log is helpful when tackling issues with unexpected behaviors or bugs. This means that developers can view the history and see why and how code has been changed before. Coupled with good practices in documentation, developers can see explanations on why code was changed in the form of commit messages. The change log is helpful when introducing new developers to the project, as they can see the history of changes and decisions that have been made.

Today's common practice in web development is to use git, which is a Distributed Version control System (DVCS). A DVCS allows easy code sharing to other developers and systems. In DVCS each party, being a client or a host, holds the full history of the project. In case the main server dies, any of the clients can restore the server back up from their version of the history. Figure 1 demonstrates the flow of the distribution of code between each party.

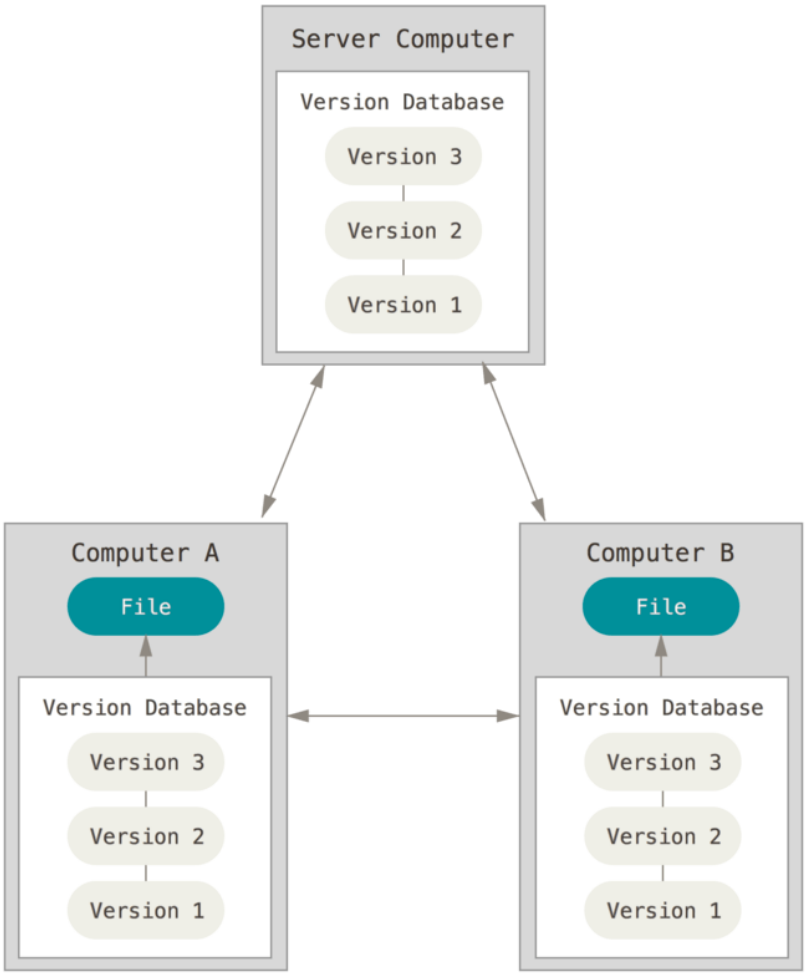

Figure 1: Distributed version control (Git, 2019)

It is possible to use a centralized version control system. They are based on the idea that there is a single central copy of the project, usually hosted on a server. Programmers will then commit their changes to this central copy. A DVCS has multiple benefits compared with a centralized version control system. It is most often faster, as pushing and pulling change sets only needs to access the hard drive and not a remote server. Multiple commits is possible to do locally before pushed to the version control.

3.2 Software Development Processes

A software process is a set of activities that leads to the production of software. The process can start from scratch or be an extension of a previous development process. Most important high-level activities include a software specification, where the main

functionalities are defined. After the specification, the software will be designed and programmed. Verification and validation follow, to ensure the software meets the customer's needs. Software maintenance is a "continuous" step that usually follows, meaning that the software is modified to meet customer's requirements changes. All these formerly mentioned tasks may contain subtasks, such as quality assurance and more specific testing.

Software process models are simplified representations on the way a software process should progress. A few known process models are the waterfall model and agile model. To compare the two, the waterfall model includes all activities in a linear order. All activities must be planned and scheduled before starting work on them. This means that a thorough knowledge on the software requirements is needed, as change is not taken into consideration. The main phases of the waterfall model are: requirements, design, implementation, testing and maintenance. However, these phases can often overlap and require information from each other. (Elgabry, 2017)

The agile method, on the other hand, is very flexible. It refers to a group of software models that are based on approaches where increments are small and typically new releases are created and released every few weeks. Agile involves customers in the development process closely and requirements can be proposed to change continuously. This can help minimize risks and extra costs that arise from faulty planning but can also pose a risk at the cost if the scope of the project is not defined. (Arsenault, 2017)

Agile works well for web development processes since they often include multiple changing requirements. These requirements can for example include the pace which technology and the market are changing. It is also said that a web developer's work is never done, as often updating and improving parts of web applications must be done continuously. Agile software development is supported by multiple software development practices. These include test and behavior driven development, which are discussed in the next two sections.

3.3 Test-driven Development (TDD)

TDD is a software development practice that can be recognized as a discipline also used outside the agile context. It focuses on repeating very short development cycles. It

combines programming, unit test writing and refactoring to achieve clean and modular code.

TDD is characterized by a set of steps known as red, green and refactor. These steps are implemented in short cycles to incrementally build up working software. The first step includes writing a failing test for a feature or assumed behavior. The test should describe the desired feature or behavior. After writing, the test should be run and verified that it displays a failure, which is often accompanied by a **red** indicator.

After making a failing test that describes the desired behavior, the simplest solution to make the test pass is implemented. The goal is to get a passing test as quickly and simply as possible. The passing of a test is often accompanied by a **green** indicator.

The last step is **refactoring**. A fast solution is not often elegant or can contain technical debt in forms of unoptimized code, for example. All issues should be cleaned up during this stage, also improving code quality. After refactoring, it is important to re-run tests and see that they still pass. (DevIQ, 2019)

There are two levels of TDD: Acceptance TDD (ATDD) and Developer TDD. ATDD, also known as behavior driven development, is an extension of TDD. It is about writing a single acceptance test and then implementing just enough production functionality to fulfil it. Figure 2 displays, how BDD and TDD work together.

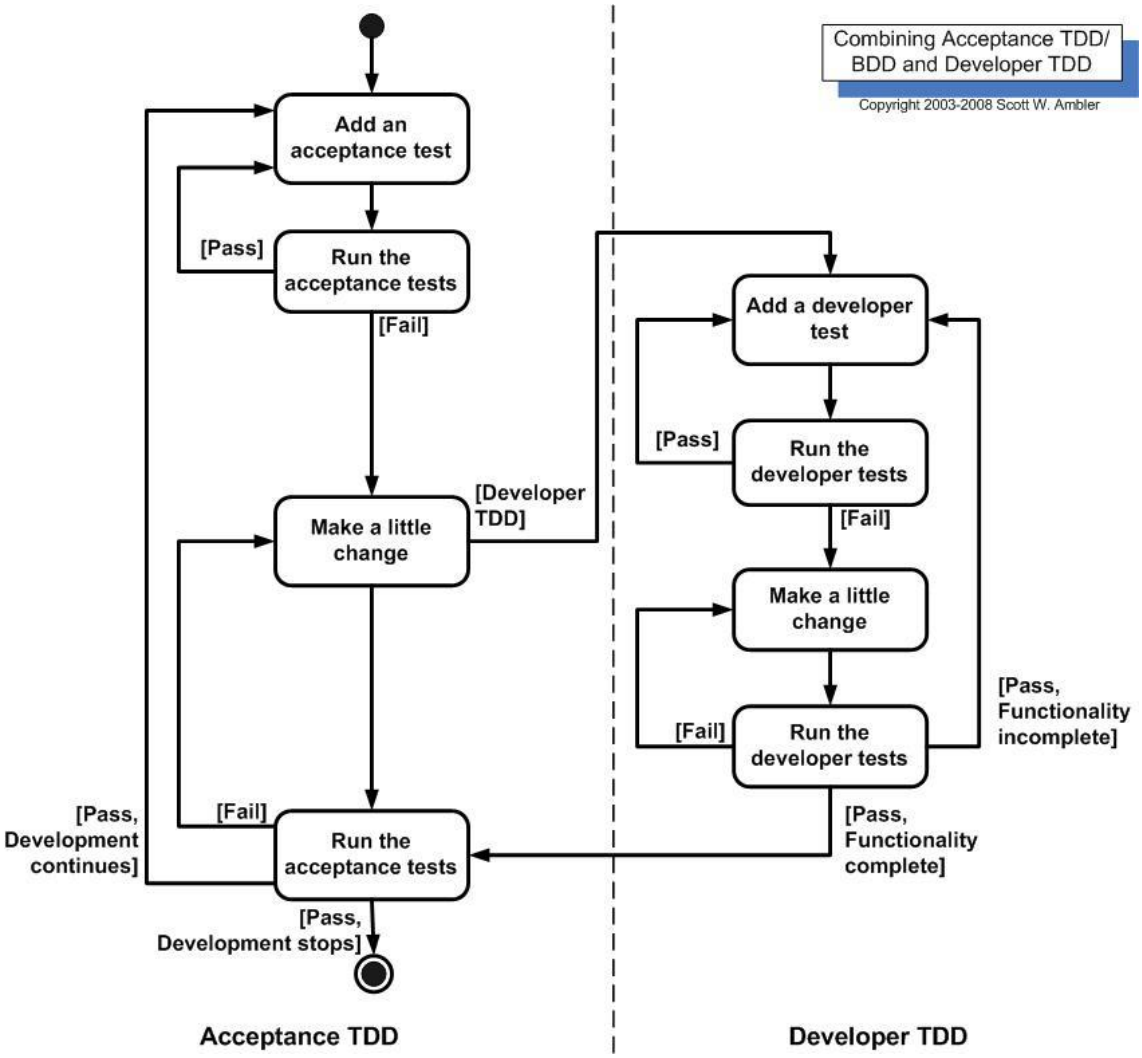

Figure 2: Introduction to Test Driven Development (Wambler, 2019)

Using BDD and TDD together, ideally one acceptance test is written, then the developer TDD approach is taken to implement the production code required to fulfill that test. This in turn requires multiple iterations through writing a test, writing a production code and getting it working. ATDD/BDD is further discussed in the next section. (Wambler, 2019)

3.4 Behavior Driven Development (BDD)

BDD is an extension of TDD. TDD starts with test code during software development, BDD starts with a desired user behavior. It also expands the user story to include scenarios that then serve as the acceptance criteria for a "passed BDD test". This can be for example a successfully executed behavior. A simplified example can be a

successful opening of a link. To spell it out as a user story, it could be “I want to be able to click the link and it opens in a separate window.”

The stories are agreed on amongst the team, product owner and QA engineers. More than in other development processes, developers require a good understanding about the business and the purpose of the product that is in development. It relies heavily on customer collaboration and common understanding about the product.

BDD tries to minify the gap between the mindsets of people who live either in the product or the solution spaces. The thought process of developers is usually more skewed toward the solution space, meaning that the code that can fulfil the requirements rather than the requirements themselves. The product owner instead might not be aware of the technical requirements of a story that might look simple. Their focus is mostly on the problem itself.

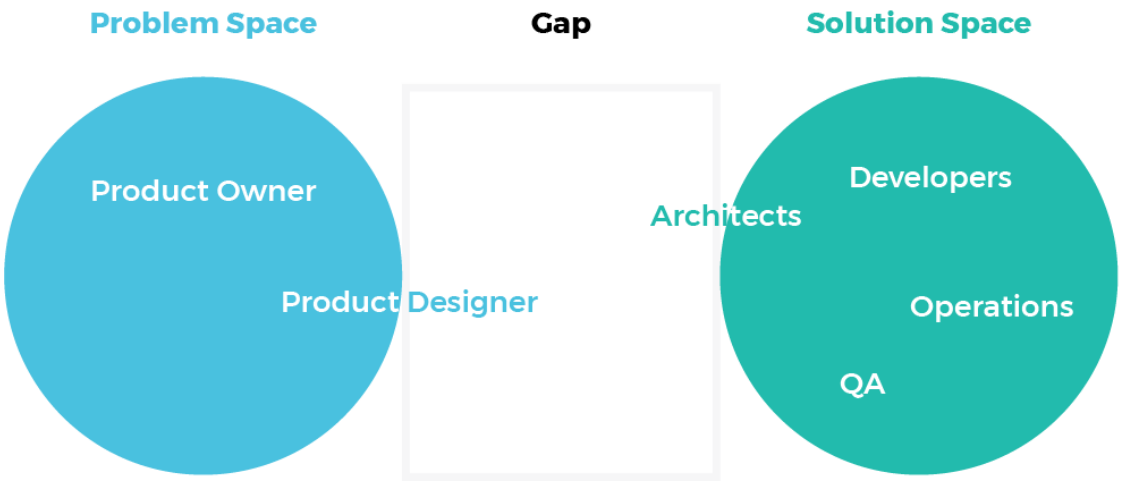

Figure 3: Behavior-Driven Development (Tharayil, 2018)

BDD is disciplined and the user stories are strictly laid out. A clear understanding of what needs to be built is achieved so that the team can assess if they have the skills required for the tasks. BDD is a good practice to consider when the problem space is complex. It is a heavy process that requires constant attention and time of all parties throughout the development process, which is why it its full potential is often not accomplished in smaller teams. (Tharayil, 2018)

3.5 The benefits of TDD

TDD allows for a faster development pace, since it provides quick feedback and less refactoring and rework are needed as detailed specifications are given in short sprints. More code can be written in a shorter time by automating what can be automated, including tests, deployments and code quality checks. The former subjects are further discussed in Chapters 4 and 5. Time is however not the only benefit to be gained from TDD.

TDD can increase the quality of the code. Automated tests can be created to evaluate the code. For example, automating the use of a prettier, which formats the code into a certain defined style, significantly lessening the cognitive load on reading the code in code reviews or in general. Combining this together with a linter, which checks for code quality rules, code quality can be significantly improved. A linter can check for various types for errors in the code and are likely to catch real bugs in the code, already during development. (Novoseltseva, 2017)

Less mistakes are made in general, due to having detailed specifications for each feature. Automated tests help in noticing issues already during development, therefore lessening bugs or possible regression in features. Developers thrive to create simplified code as it is easier to test, and the code is planned beforehand. (Novoseltseva, 2017)

A good workflow is not only beneficial to the customer, but it can significantly increase the developer experience as well. Unifying project workflows across teams will help developers to adapt faster when switching between projects. Workflows should be adopted based on the needs of the team and the customer. There is no workflow that fits all projects.

4 AUTOMATIC CODE SHIPPING

A collection of the technical practices introduced in the following sections are a cause of the agile and lean methodologies. Agile is a project management method that helps teams provide quick and unpredictable responses to the feedback they receive in a project. (Gonçalves, 2019). Few of the main principles of agile are to deliver customer satisfaction by delivering software continuously and to deliver it within a shorter timescale.

It is said, that agile is not agile without continuous delivery. For agile to work, all phases of the development lifecycle should be agile as well. (Buchanan, 2019). The following sections dive deeper into practices and methods that follow the idea of agile, which are for example delivering software quick and continuously.

4.1 DevOps

DevOps is a set of practices, that automates the processes between software development and IT teams, in order that they can build, test, and release software faster and more reliably. It can be referred to as a culture or a philosophy that emphasizes on the collaboration between teams that have previously functioned separately, such as the organizational level and the development team (Atlassian, 2019).

Closer collaboration is the primary characteristic of DevOps. Closer collaboration requires for an attitude of shared responsibility. “If a development team shares the responsibility of looking after a system over the course of its lifetime, they are able to share the operations staff’s pain and so identify ways to simplify deployment and maintenance (e.g. by automating deployments and improving logging).” (Wilsenach, 2015) The same applies for the operations staff, as they are able then to work closer with developers to better understand the operational needs of a system and to meet them. The need can include for example adopting new automation tools and practices.

Close collaboration requires other cultural shifts to be made in the team culture and organizational culture. Figure 4 displays the essential cultural practices that should be in place for DevOps culture to work.

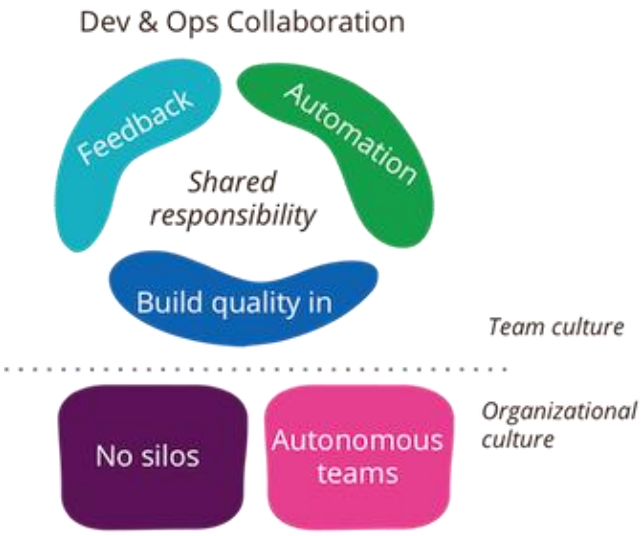

Figure 4: DevOps Culture (Wilsenach, 2015)

To properly execute DevOps, automation needs to be properly considered and put into effect, which is why practices such as CI and CD should be included in the software development process. These practices are introduced in the following sections.

4.2 Deployment Pipeline

To better understand CI and CD, this section describes the deployment pipeline where these strategies are introduced. The deployment pipeline consists of local tests, such as verifying that the code builds and works locally, then committing into version control. After commit, the code moves into the code review state. Code review can be part of the deployment pipeline, or it can be done before the code goes into the pipeline. After a positive assessment of the code, changes are merged into the mainline branch.

After accepted, the mainline branch goes into the deployment pipeline. The build server takes the code, compiles and builds it. Every pipeline consists of different parts and it can be configured according to the team's or software needs. It can consist of tests and different validation steps. These can be for example a unit, an integration or end-to-end tests. Testing methods are discussed in more detail in Chapter 5.

At the end of the pipeline, there is the deployment step. If continuous deployment strategy is followed, this step is also automated. The deployment can also require manual verification. In this case, the pipeline is referred to as continuous delivery. Figure 5 shows steps of the pipeline and demonstrates stages that are automated in the CI, CD and continuous delivery workflow strategies.

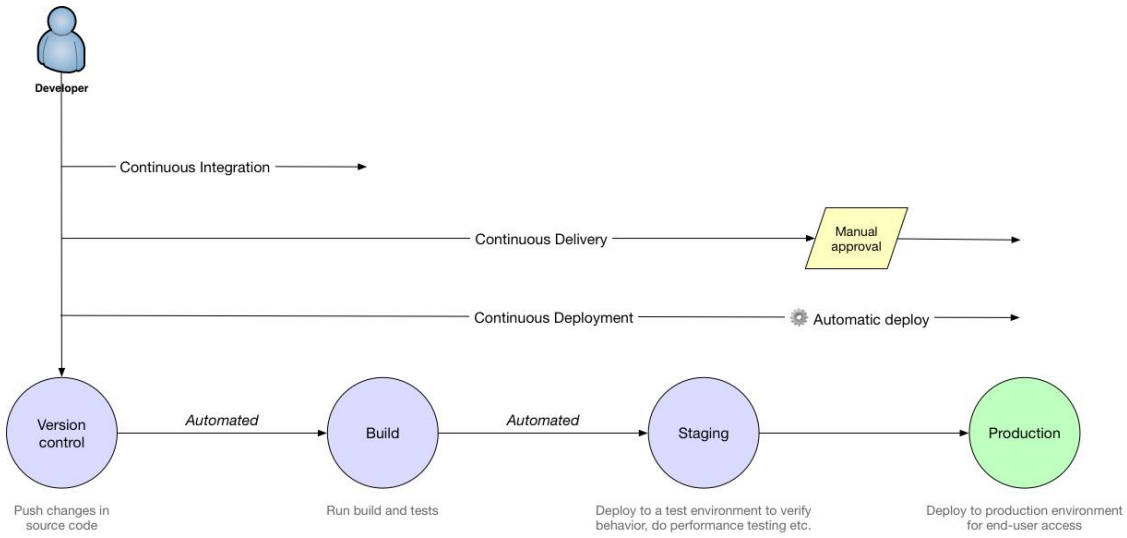

Figure 5: Continuous delivery workflow strategies (Anastasov, 2017)

Problems that CI and CD solve or aim to solve are:

- Uncertainty of committed code in the mainline branch
- Instability in code
- Increased number of bugs that are found long after releasing, possibly tangled with each other
- Long release times, as the code needs to be manually stabilized before the release and causes wait times

(Buchanan, 2019).

4.3 Continuous Integration (CI)

CI is a workflow strategy that helps ensure that each team member's changes will integrate with the current version of the project. This reduces merge conflicts, helps catch bugs and can improve code quality. CI consists of short-lived feature branches that are

merged to the mainline branch in quick iterations, preferably multiple times a day. (GitHub, 2017)

Team will practice CI in conjunction with automated testing, using a dedicated server or CI service. Whenever a developer adds new work to the branch, the server will automatically build and test the code to determine whether it works and can be integrated with the mainline branch. The tests can include for example unit tests, performance and UI behavior tests. Testing is further discussed in Section 5. Code styling and linting can also be run during the build process for ensuring good code quality. (GitHub, 2017)

CI is not necessarily faster shipping code. The deployment is still done manually, and due to good practices, mostly there is in addition a manual review process. This means that every line that is merged will be reviewed by another developer than the maker. There are many benefits in the manual review process. It will help to reduce logical errors and bugs, keep code quality high and to make sure there are enough tests and that they are of good quality. A manual review process also ensures that the team stays on board with the state of the project code. It enables a space and time for asking questions about the code and why different approaches were taken. Since the idea in CI is to keep the features short and quick, the review process does not introduce much overhead. (GitLab, 2019)

CI can face multiple challenges in project teams. The source can often be developers themselves. For instance, developers might see CI as counterproductive as it can take time from actual development. This kind of attitude can also result in ignoring error messages and broken builds. It can be done purposely, but also the amount of notifications from CI can become overwhelming, causing the developers to mute the notifications and miss the important updates as well. (GitLab, 2019)

CI can also be hard to adopt, since getting used to the daily building and testing can prove to be challenging. Fears of breaking the build or not passing the tests and therefore missing deadlines can also raise pressure amongst the team. It is important to emphasize that failing tests are mostly a good thing, since a bug that would have otherwise reached the user, is realized early and can be assessed.

4.4 Continuous Delivery (CD)

CD is an extension of CI, as it adds the automated deployment step that CI doesn't include. Its purpose is on releasing changes to customers faster. Before introducing CD to the project, CI practices should be in good order and already in practice. (GitLab, 2019)

The timing of the deployment is a manual, but the whole deployment process is automated. The deployment should be relatively risk-free as rolling back to the previous version should be easy. It is also possible to switch through previous versions with just a click. CD does not dictate how often you deploy code to production and setting the deployment is a manual. (GitLab, 2019)

The pipeline is multiple steps which the code runs through until it ends in a deployment step. The pipeline manages code in that timeframe, it then is deployed somewhere. To summarize, continuous delivery is the practice of developing software in such a way that it could be released at any time. (GitLab, 2019)

4.5 Continuous Deployment

Continuous deployment relies on having a pipeline of automated steps to deployment, meaning that it is also an extension of Continuous Delivery (CD). CD means that the ability is there, but human dictates when the deployment is made. Continuous deployment instead means that every change which lands in the mainline branch is pushed to production automatically, with no human interaction. Errors during tests interrupt the deployment, which then needs a manual review. (GitLab, 2019)

Continuous deployment requires high quality testing to make sure that each change is ready for production. It accelerates the release and feedback cycle of the end-user, however the risk of having completely automatized deployment is often too high for regular use. These risks can include downtime and unfinished features if the development workflow is not strictly implemented. Therefore, in most cases deployment still is best done by human interaction. (GitLab, 2019)

AUTOMATED TESTING

This section focuses on test automation methods. Tests can be simplified as scripts that run against features or code that is implemented. There is a way to run through various scenarios to check that the results produced by tests are the same as expected. They help raise red flags where necessary.

Automated tests can be ran by a computer, meaning that thousands of scenarios can be checked in a matter of seconds. The main benefits of test automation are to help ensure a level of quality and to detect bugs before they reach users. Broken software costs time and money. Continuous shipping of defects also makes the software increasingly hard to maintain for developers, costing even more money (Startup Lab, 2019)

There are three main approaches, which are a unit, integration and end-to-end testing. Figure 6 conveys what amount of resources each approach uses.

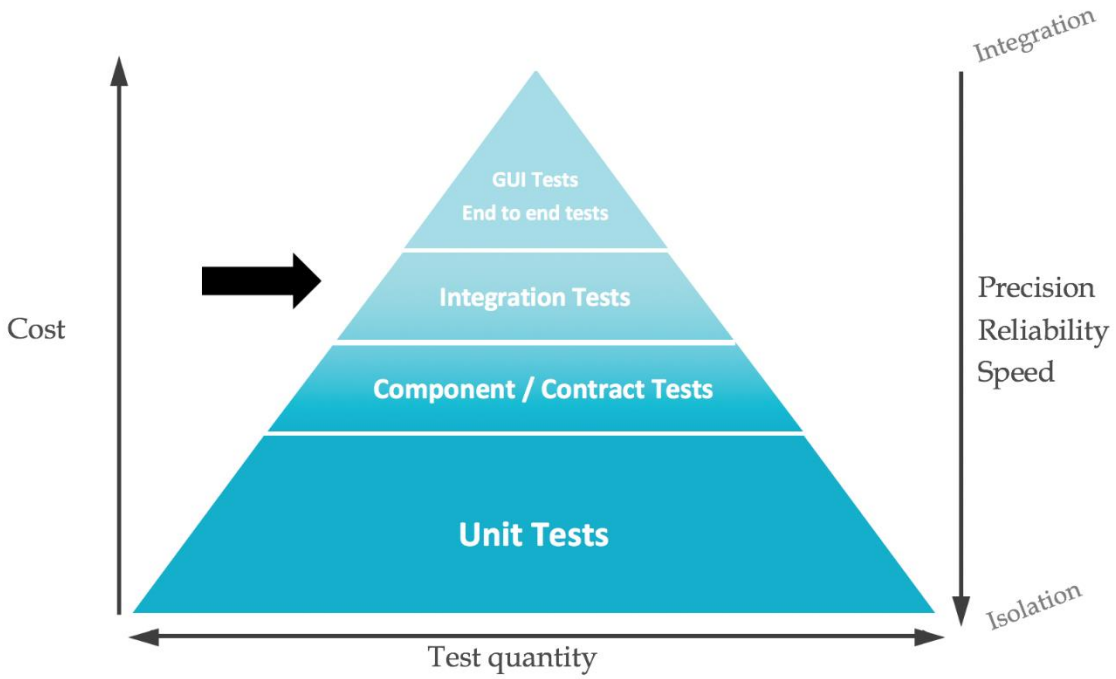

Figure 6: The Test Pyramid in Practice (Gillispie, Houssein, Linden, 2018)

Formerly mentioned testing approaches are often implemented in various ratios and there are no general guidelines, however “Google suggests a 70/20/10 split: 70 % to unit tests, 20 % to integration tests, and 10 % to end-to-end tests” (Wacker, 2015). This

suggestion will be further investigated in the following sections, with examples from each testing approach.

4.6 Unit Tests

Unit tests are isolated tests of individual pieces of code. They are a way to ensure that the code works in its standalone environment. This code can, for example, be a JavaScript component. It can also be a test of a class or utility function. (Startup Lab, 2019)

The purpose of unit tests is to validate that each smallest part, unit, of the application performs as designed. They will determine whether a returned value equals the value that was expected when the feature was written. In TDD, the value that will be expected when the code is written. They are often run after every change to the source code and can be run automatically depending on continuous integration and delivery pipeline configurations.

Unit tests can be written with for example Jest, which is a JavaScript testing framework. It provides all necessary tooling to create and run unit tests. Figure 7 displays an example of a simple unit test of a hypothetical function multiply.

```
1   // Multiply module (multiply.js)
2   function multiply(a, b) {
3     return a * b;
4   }
5   module.exports = multiply;
6
7   // Test output of the multiply module (multiply.test.js)
8   const multiply = require('./multiply');
9
10  test('adds 2 * 2 to equal 4', () => {
11    expect(multiply(2, 2)).toBe(4);
12  });
```

Figure 7: Unit test with Jest

When writing React components, part of the unit testing can be implemented with **snapshots**. Snapshots help make sure that the user interface does not change

unexpectedly. It compares new code with a snapshot that was previously generated from the component output. If the code differs, the test will fail. (Jest, 2019)

Figure 8 describes a typical flow for snapshot testing: The expected output is saved as a JSON tree. A difference check runs between the new and previous snapshot and if there are no differences, the test passes. (Vieira, 2017)

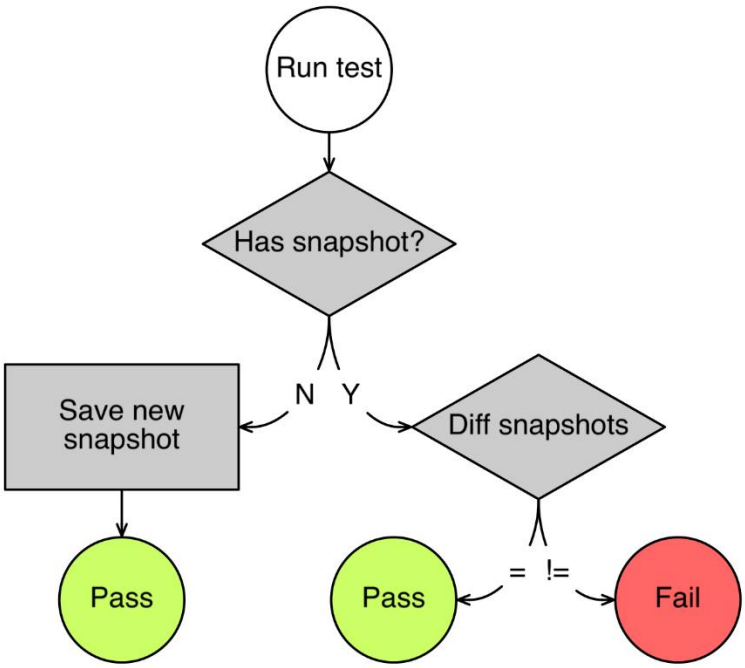

Figure 8: Snapshot testing with Jest framework (Vieira, 2017)

Snapshots should not be taken as the only source of truth as they are just a way to quickly check that nothing changes. The original snapshot test can easily be false as they are automatically created, and the code is not verified by the developer. They should be used together with other testing methods for an accurate result.

4.7 Integration Tests

Integration tests are more complex than unit tests, but simpler than E2E tests. Scripts can be used in the development of new features to ensure they are not breaking functionality in other areas. It simply ensures that units, tested with unit tests, work together as expected.

For React applications, integration tests are used to test the interactions between separate components. This includes testing interactions, which are typically performed with calling prop functions, such as an onClick event. It also includes a testing of manipulation of the component state and direct manipulation of the DOM in React lifecycle methods.

To test the formerly mentioned interactions, it is required to render the component within a functioning DOM. This is possible with a few JavaScript testing libraries, such as Enzyme and Jest. Figure 9 displays an example of verifying the DOM manipulation of a component in React.

```
2   // Copyright 2004-present Facebook. All Rights Reserved.
3
4   import React from 'react';
5   import ReactDOM from 'react-dom';
6   import * as TestUtils from 'react-dom/test-utils';
7   import CheckboxWithLabel from '../CheckboxWithLabel';
8
9   it('CheckboxWithLabel changes the text after click', () => {
10    // Render a checkbox with label in the document
11    const checkbox = TestUtils.renderIntoDocument(
12      <CheckboxWithLabel labelOn="On" labelOff="Off" />
13    );
14
15    const checkboxNode = ReactDOM.findDOMNode(checkbox);
16
17    // Verify that it's Off by default
18    expect(checkboxNode.textContent).toEqual('Off');
19
20    // Simulate a click and verify that it is now On
21    TestUtils.Simulate.change(
22      TestUtils.findRenderedDOMComponentWithTag(checkbox, 'input')
23    );
24    expect(checkboxNode.textContent).toEqual('On');
25  });
```

Figure 9: Integration test with Jest (Jest, 2019)

4.8 End-to-end (E2E) Tests

E2E testing is where the whole application is tested together. The idea is to simulate the behavior of a real end-user using the application. This means that the testing setup needs to be much more complex. For creating the test cases, there are headless browsers, which are browsers without any graphical interface. Headless browsers make it easier to run test cases is automated testing environment.

For JavaScript applications, Puppeteer can be used together with formerly mentioned Jest to mimic human-like interaction with the browser. It can capture snapshots and generate PDFs of screens, which makes it a good tool for visual-based testing. (2019, Puppeteer)

5 PROOF OF CONCEPT

A proof of concept project was implemented to further demonstrate points made in theoretical sections. The proof of concept included a PostgreSQL relational database with two resources. A RESTful API was created to act with the database to support CRUD operations (Create, Read, Update, Delete). The backend used NodeJS and Express for handling HTTP requests.

A frontend application was created with React using TypeScript. It includes a unit tests for components and integration tests for testing interaction between them. The original plan was to implement E2E tests as well, but time constraints did not allow this.

The application was built on Docker to enable the use of GitLab's built-in CI/CD tools and to ease the development process. Custom Docker images were created for the frontend and backend. The database uses PostgreSQL's own official Docker image, which has been configured.

A continuous delivery pipeline was implemented using GitLab's built-in CI/CD tools. Every merge request trigger automated tests and the automatic generation of a branch-based environment. This means that each active branch has its own environment for visual feature verification. The tests and the pipeline are further explained in the following sections.

5.1 Creating tests

For a starter project such as this, creating unit tests is very simple. In real-world applications, unit tests are more difficult to implement and require a thorough planning of the feature. As there were only a few components interacting with each other, integration tests were relatively simple to create. These together create a good base for the development of an application.

Unit tests can and should be used beyond React applications as well. In this case, they were used to do the snapshot tests of the response from the API. This was done by making an API call against a real server and snapshot testing the result with Jest. With passing API snapshot tests, it is easy to see that the user interface and backend work

together as expected. For example, a simple API test would be to test a login or a create feature, which Figure 10 demonstrates.

```
import * as API from 'api';

test('failed login (bad password)', async () => {
  let data;
  try {
    data = await API.login('me@example.com', 'wrong_password');
    fail();
  } catch(e) {
    expect(e.response.data.error).toMatchSnapshot();
  }
});

test('failed login (bad username)', async () => {
  let data;
  try {
    data = await API.login('not-a-real-account@example.com', 'password');
    fail();
  } catch(e) {
    expect(e.response.data.error).toMatchSnapshot();
  }
});

test('good login', async () => {
  try {
    const response = await API.login('test-account@example.com', 'supersecret!');
    expect(response).toMatchSnapshot();
  } catch(e) {
    fail();
  }
});
```

Figure 10: Snapshot Testing APIs with Jest (Ceddia, 2017)

E2E tests were not yet implemented since they require a lot of effort to implement. They can also bring some issues, as they can be slow and error-prone. Errors can be caused by timing bugs. Also, as E2E tests are extremely specific, snapshots provide a less opinionated starting point for testing.

5.2 Building continuous pipelines

CI/CD pipelines were configured using the .gitlab-ci.yml YAML file. It defines the structure, the order of the pipelines, determines what to execute using the GitLab runner and what decisions to make when specific conditions are encountered. (GitLab, 2019). Creating YAML configuration files is difficult at the start, since almost all parameters require knowledge of the GitLab pipeline concepts. Figure 11 displays one of the first

versions of a configured .gitlab-ci.yml file. It is configured to automatically run tests for API and the database and to cache the node_modules folder between runs.

```
image: node:latest

services:
  - postgres:latest

cache:
  paths:
  - node_modules/

test_api:
  script:
  - yarn install
  - yarn run test ./tests/api.test.ts

test_db:
  script:
  - yarn install
  - yarn run test ./tests/db-postgres.test.ts
```

Figure 11: Configured GitLab CI file

After the pipelines are configured, the pipelines can be seen in GitLab like Figure 12 displays. All commits to be configured to run the pipeline and the status of all pipelines are displayed as follows.

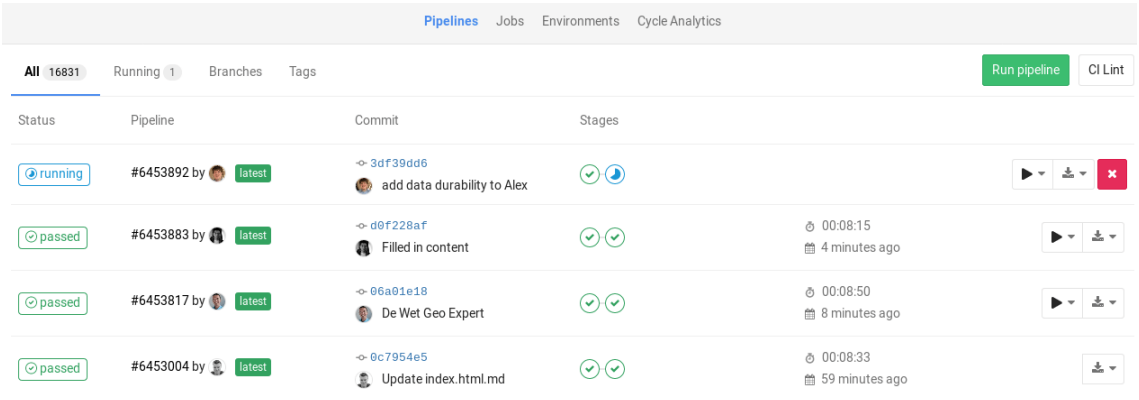

Figure 12: Pipelines in GitLab (GitLab, 2019)

6 CLOSING CHAPTER

The need for good development processes and automation tools is strongly driven by the agile movement. In an agile environment, requirements evolve quick and the need for the quick shipments of bug-free features increases. This is where agile methods such as TDD and DevOps are vital.

TDD is a practice that can be difficult to adopt. It requires a mindset shift to developers and whole organizations. Using more time on tests than actual development can feel like daunting and can cost a lot of money at first. In smaller organizations, TDD can also entirely be on the responsibility of the developer as there might be no architects or QA engineers to help with writing tests.

After adopting and getting used to TDD principles, the development process is significantly more effective. In addition to increasing code quality, applications are more stable and fewer bugs reach the end-user. An increased user experience often translates directly to more money.

For DevOps, there are a few parts that clearly work as its base. These include CI/CD, cloud infrastructure and test automation. This thesis touched on CI/CD and test automation. The main takeaway from the thesis was that a good workflow benefits the customer in regard to time and money but also adds to the developer experience in multiple ways.

CI/CD increases developer confidence to implement new features and push updates quickly, even in large-scale applications. This is due to improved code validation by for example catching bugs at an early stage. A good CI/CD pipeline also lessens the time used on communication between team members as all members are aware on the changes in the codebase.

In general, manual processes should be avoided and tools should always be the main source of knowledge in teams, not people (Armstrong, 2019). It is not possible to automate everything, such as code reviews, but aim to make them as effortless and fast as possible by using tools to eliminate problems already during development.

Personally, the interest for this thesis topic came from understanding the need for better development processes for web applications. Today, web applications are complex.

Instead of static HTML-pages, they are becoming more like native applications. Unlike native applications, using web technologies, it is possible to implement applications that do not depend on any specific platforms.

One codebase for all platforms saves a lot of money and time. Especially now that there are ways to create native-like web applications, it is safe to assume their popularity will continue to grow in the future. Therefore, it is increasingly important to adapt suitable development principles, to create more reliable applications with impressive user experience. These principles should, however, be always picked on a team basis as there is not one solution that fits all.

REFERENCES

Anastasov, M. (2017). What's the Difference Between Continuous Integration, Continuous Deployment and Continuous Delivery? Available at: https://semaphoreci.com/blog/2017/07/27/what-is-the-difference-between-continuous-integration-continuous-deployment-and-continuous-delivery.html [Accessed 25 Mar. 2019].

Atlassian. (2019). DevOps: Breaking the Development-Operations barrier. Available at: https://www.atlassian.com/devops [Accessed 18 Mar. 2019]

Armstrong, P. (2019). Move fast with confidence. [video]. Available at: https://www.youtube.com/watch?v=ikn_dBSski8 [Accessed 29 May 2019].

Arsenault, C. (2017). Agile Web Development – a Comprehensive Overview. Available at: https://www.keycdn.com/blog/agile-web-development [Accessed 22 May 2019].

Ceddia, D. (2017). Snapshot Testing APIs with Jest. Available at: https://daveceddia.com/snapshot-testing-apis-with-jest/ [Accessed 20 May 2019].

Gonçalves, L. (2019). What is Agile Methodology. Available at: https://luis-goncalves.com/what-is-agile-methodology/ [Accessed 15 May 2019].

DevIQ. (2019). Test Driven Development. Available at: https://deviq.com/test-driven-development/ [Accessed 20 May 2019].

Electron. (2019). Electron Documentation: About Electron. Available at: https://electronjs.org/docs/tutorial/about [Accessed 24 Mar. 2019].

Elgabry, O. (2017). Software Engineering – Software process and Software Process Models Available at: https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc [Accessed 15 Mar. 2019].

Wilsenach, R. (2015). DevOpsCulture. Available at: https://martinfowler.com/bliki/DevOpsCulture.html [Accessed 18 Mar. 2019].

Gillispie, L., Houssein, A., Linden, L. (2018). The Test Pyramid in Practice. Available at: https://blog.octo.com/en/the-test-pyramid-in-practice-1-5/ [Accessed 23 Mar. 2019].

Git. (2019). Getting Started About Version Control. Available at: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control [Accessed 25 Feb. 2019].

GitHub. (2019). GitHub's take on CI: move quickly not recklessly https://resources.github.com/whitepapers/continuous-integration-github [Accessed 15 Mar. 2019]

GitLab. (2019). GitLab CI/CD. Available at: https://docs.gitlab.com/ee/ci/ [Accessed 15 May 2019].

GitLab. (2019). Introduction to CI/CD with GitLab. Available at: https://docs.gitlab.com/ee/ci/introduction/ [Accessed 15 May 2019].

Google Developers. (2019). Progressive Web App Checklist. Available at: https://developers.google.com/web/progressive-web-apps/checklist [Accessed 24 Mar. 2019].

Jest. (2019). Snapshot Testing. Available at: https://jestjs.io/docs/en/snapshot-testing [Accessed 12 Apr. 2019].

Kostecky, I. (2019). An Approach to Software Deployment: Continuous Integration Practices. Available at: http://urn.fi/URN:NBN:fi:amk-201902041935 [Accessed 2 May 2019].

Nilan, Y. (2018). Introduction to Continuous Integration & Continuous Deployment. Available at: https://medium.com/pulseque/introduction-to-continuous-integration-continuous-deployment-38c3ebc07221 [Accessed 18 Apr. 2019].

Novoseltseva, E. (2017). 20 Advantages of Test Driven Development. Available at: https://apiumhub.com/tech-blog-barcelona/advantages-of-test-driven-development/ [Accessed 20 May 2019].

Open Source Initiative. (2019). Licenses & Standards. Available at: https://opensource.org/licenses [Accessed 24 Apr. 2019].

Parshina, M. (2018). JavaScript Beyond the Browser. Available at: http://urn.fi/URN:NBN:fi:amk-2018061113509 [Accessed 29 Apr. 2019].

Puppet. (2019). Embrace DevOps, or get left behind. Available at: https://puppet.com/resources/whitepaper/2015-state-devops-report [Accessed 16 Feb. 2019].

Puppeteer. (2019). Puppeteer. Available at: https://pptr.dev/ [Accessed 28 May 2019].

Red Had. (2019). What is open source? Available at: https://opensource.com/resources/what-open-source [Accessed 24 Apr 2019].

Tharayil, R. (2018). Behavior-Driven Development: Simplifying the Complex Problem Space Available at: https://www.solutionsiq.com/resource/blog-post/behavior-driven-development-simplifying-the-complex-problem-space/ [Accessed 10 Apr.2019].

The Startup Lab. (2019). What is Automated Testing? Available at: https://www.youtube.com/watch?v=Nd31XiSGJLw [Accessed 24 Mar. 2019].

Wacker, M. (2015). Just say no to more end-to-end tests. Available at: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html [Accessed 23 Mar. 2019].

Wambler, S. (2019). Introduction to Test Driven Development. Available at: http://agiledata.org/essays/tdd.html [Accessed 28 May 2019].

Vieira, L. (2017). Snapshot testing React components with Jest. Available at: https://medium.com/@luisvieira_gmr/snapshot-testing-react-components-with-jest-3455d73932a4 [Accessed 15 Apr. 2019].

VT Netzwelt. (2018). Native app development vs hybrid app development. Available at: https://www.vtnetzwelt.com/mobile-application-development/native-app-development-vs-hybrid-app-development/ [Accessed 24 Mar. 2019].