

Fachhochschule der Wirtschaft
-FHDW-
Mettmann

Master's Thesis

Topic:

**Conception, Evaluation and Prototyping of a
Distributed Patch Deployment and Control System**

Examiner:

Dr. rer. Nat. Dr. Markus Borschbach

Secondary Examiner:

Prof. Dr. Philipp Rohde

Author:

Steven Datzmann

Suitbertusstraße 20

40223 Düsseldorf

Matriculation Number: 9152264

IT-Management and Information Systems

Date of submission:

08.05.2017

Executive Summary

Starting with the industrialization, process automation is key to strengthen a company's competitive ability on the market for manufacturing businesses until today. But it is strangely enough often left aside in the software sector. Many tasks in maintaining software solutions are still conducted more or less manually and are prone to simple mistakes which could have been avoided by reducing human interaction. This leads to a loss in productivity and a degradation in software quality for supplying maintenance.

Against this background, automation of repetitive, laborious and error-leaden processes is a crucial task to increase competitive ability and providing professional maintenance to software solutions.

This thesis develops a concept for automating the vital process of patch deployment based on a case study. The case study describes the manual process of creating, delivering and installing patches which are currently in use. Out of this case study, key goals are derived which would aid in reducing or outright eliminate errors in the process.

Guided by the limited, existing, research on automating this part of the software maintenance process, existing solutions are examined. After evaluation of these solutions, implementation into a generalized, streamlined workflow concept is discussed.

Results of this work include a prototype which partially automates the processes of patch creation and fully automates delivery and installation. Also, the impact of using such a solution is discussed in regard to a reduction in necessary effort to maintain software from a software suppliers and customers view.

Table of Contents

Executive Summary.....	II
Table of Contents	III
List of Tables	V
List of Figures.....	VI
List of Abbreviations	VIII
1 Introduction.....	1
2 Case Study	2
2.1 Infrastructure.....	3
2.2 Workflow.....	4
2.3 Problem and Goal Definition	7
3 Elementary Concepts	9
3.1 Patch Deployment	9
3.2 Deployment Pipeline	11
3.3 Hashing	14
3.4 Compression.....	16
4 Evaluation of existing solutions	20
4.1 Filesystem	20
4.2 Git.....	23
4.3 Subversion.....	27
4.4 Database	30
4.5 Database Migration Tools	32
5 Conception	34
5.1 Architecture and Platform Choice.....	34
5.2 Process Modelling	36
5.3 Integration of Solutions	42
5.4 Data model	44
6 Prototyping.....	48
6.1 Delta Analysis.....	50
6.2 Create Patch Package	56
6.3 Installation	61

7	Testing.....	66
7.1	Boundaries and Limitations.....	66
7.2	Results.....	69
8	Estimated Impact	71
8.1	Quantitative Impact.....	71
8.2	Qualitative Impact	73
9	Conclusion	75
9.1	Summary	75
9.2	Outlook	77
	Appendix	78
	List of Cited Literature.....	82
	Ehrenwörtliche Erklärung	86

List of Tables

Table 1: Character Checksum Example.....	14
Table 2: Character SHA-1 Example	15
Table 3: Huffman coding symbols.....	18
Table 4: Performed changes to example database	30
Table 5: OSC Table Change Phases Overview	33
Table 6: Patch Table.....	49
Table 7: Connector Application State.....	49
Table 8: MySQL “SHOW FULL COLUMNS” Result:	53
Table 9: Connector Create Patch Form	56
Table 10: Tests	67
Table 11: Test Results	69
Table 10: Patch Effort:	71
Table 11: Patch Effort Adjusted:	72

List of Figures

Figure 1: LAMP Stack:	3
Figure 2: Patch Workflow:	4
Figure 3: Patch Workflow:	7
Figure 4: Prototyping Software Life cycle:	9
Figure 5: Continuous Delivery Promotion:	12
Figure 6: Deflate compression block:	17
Figure 7: Huffman tree:	19
Figure 8: Basic filesystem version example:	20
Figure 9: Advanced filesystem version example:	21
Figure 10: Git file Status:	25
Figure 11: Git remote repository:	26
Figure 12: Subversion repository:	27
Figure 13: Subversion locks:	28
Figure 14: Basic database version example:	30
Figure 15: Patch Library:	34
Figure 16: Patch Connector:	35
Figure 17: Filesystem Delta Analysis:	37
Figure 18: Database Delta Analysis:	38
Figure 19: Patch Package Contents:	40
Figure 20: Patch Installation Workflow:	41
Figure 21: Patch Structure XML Example:	44
Figure 22: Patch Structure JSON Example:	45
Figure 23: Patch Structure JSON Example Extended:	46
Figure 24: Patch Structure XML Example Files:	47
Figure 25: Connector Client - GUI:	48
Figure 26: Connector Frontend - getDelta:	50
Figure 27: Connector Backend – File Listing:	50
Figure 28: Connector Backend – File Delta:	51
Figure 29: Connector Backend – Database Listing:	52
Figure 30: Connector Database Structure Example:	54
Figure 31: Connector Backend – Database Delta:	54
Figure 32: Connector Backend – Delta Removed:	55
Figure 33: Connector Backend – Delta Result:	56
Figure 34: Connector Frontend – Show Database Changes:	57
Figure 35: Connector Frontend – Store:	57
Figure 36: Connector Backend – Create Patch Package: Files	58
Figure 37: Connector Backend – Create Patch Package: Database	58
Figure 38: Connector Backend – Create Patch Package: Structure	59
Figure 39: Connector Backend – Create Patch Package: Metadata	59
Figure 40: Library – Store Endpoint	60

Figure 41: Connector Backend – File Backup:	62
Figure 42: Connector Backend – Table Preparation	64

List of Abbreviations

CPU	Central Processing Unit
CVS	Concurrent Versions System
DB	DataBase
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
LAMP	Linux / Apache / MySQL / PHP
LHM	Large Hadron Migrator
MD	Message Digest
MIT	Massachusetts Institute of Technology
OS	Operating System
OSC	Online Schema Change
PDF	Portable Document Format
PHP	PHP Hypertext Preprocessor
SHA	Secure Hashing Algorithm
SQL	Structured Query Language
URL	Uniform Resource Locator
VCS	Version Control System
XML	Extensible Markup Language

1 Introduction

It is an accepted fact, that repetitive processes performed by human beings bear the probability of human error. This error leads to a loss of efficiency for the performed process which then results in further degradation of productivity, due to necessary corrective measures which have to be taken in order to rectify said error. Companies tend to minimize errors and thus increase productivity through simplifying and when possible, automating processes to a point where human interaction is only needed for making decisions, which cannot or must not be done by the program.

This thesis will focus on optimizing the vital process of patch deployment for software products to a point where fragile, repetitive parts are solely performed by a prototyped, specialized software. The only choices left to a human operator should be where and when a patch will be deployed, not which technical measures need to be taken to do so. This thesis will be guided by already existing, similar, solutions to parts of the actual problem at hand and incorporate “state-of-the-art” methods to solve common issues of patch deployment.

2 Case Study

This work focuses on solving a given problem based on a specific case study and generalizing it to provide the solution for patch deployment and control to a wider audience than just a single case. The following describes the application infrastructure and the workflow used to create and distribute patches for the application.

The case study's application is an information portal which is used by banks to facilitate communication with the customer. As the portal is used for the specialized financial product factoring, which basically means selling invoices to a bank to increase liquidity, there is a high need for constant communication with each customer. Customers need to be able to upload their invoices reliably and see their invoices status. They must be able to check their financial situation and related data in the portal application. Due to the financial nature of displayed information, there is no room for incomplete and/or corrupted data.

Per the case study, it is assumed that the application can be used as a hosted service via the software supplier as well as an application which can be installed on the client's systems. In the hosted service scenario, the software supplier hosts the application environment for a customer and is also in charge of keeping their installation up to date. In the client hosted scenario, the client's systems have at least a two-stage process of installation. First, the patch is installed on the test system, where the new functions, as well as the previously deployed functions, are tested by the client's test team. After all tests have been concluded positively; the patch is then installed on the productive system.

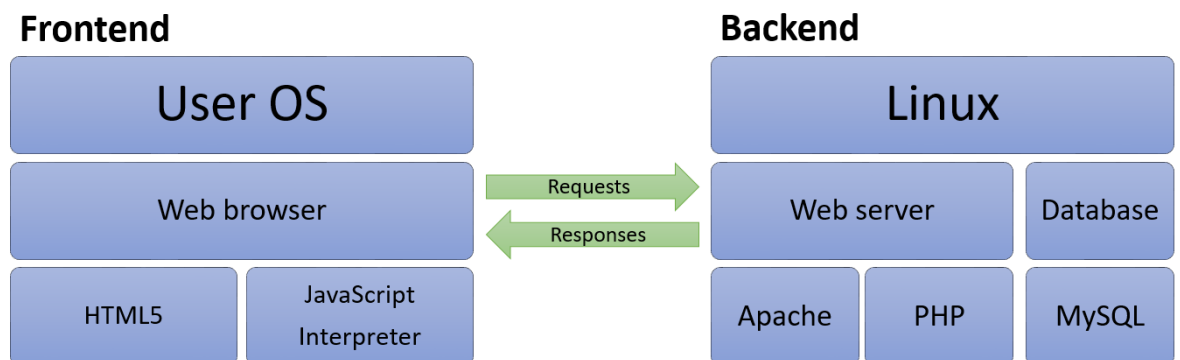
Installation to the productive environment is a business-critical task as it usually involves a maintenance time window in which the application is not in a stable state and thus not useable by its users. Furthermore, the entirety of the later on described workflow is conducted manually. There are no parts which have been automated thus far and as such there is a high probability of human error. This makes it especially critical, as no rollback mechanisms are in place, rollbacks are also conducted manually.

2.1 Infrastructure

The software system that is taken into consideration is based on an HTML5 frontend which is deployed to the user's browser. The deployment of the applications frontend is performed by requesting the application installations URL with a browser and automated through a PHP backend which serves the required files. The backend system which creates the mentioned HTML5 frontend also performs other requested actions, like retrieving, storing and processing data. This backend installation is the main component of the application and thus needs a reliable way of patch deployment.

The PHP backend is installed on an Apache webserver running on a Linux system as an application server. For persistent storage of data, a MySQL database is used. This database is installed on the same host or connected to the Apache host. In general, this is considered a classic LAMP¹ stack environment:

Figure 1: LAMP Stack:



Source: Own illustration based on Dougherty (2001)

This application infrastructure is widely used for internet applications. Per Netcraft, a grand total of about 52 percent of websites on the internet, which were part of Netcraft's analysis used an Apache server and thus a similar setup in 2014.² Also, the usage of MySQL as a persistent storage solution is widely adapted as it is the second most used database per DB-Engines Ranking.³ These facts underline the possibility to adapt further results from this thesis to other case studies, as the used infrastructure is not a special, but a common one.

¹ LAMP stands for Linux, Apache, MySQL, PHP (or Pearl).

² See Netcraft (2014).

³ See DB-Engines (2017).

2.2 Workflow

In the considered case study the general workflow for creating and distributing a patch can be divided into three parts, where each part is dependent on the previous one (see Figure 2):

- **Patch Construction:**

Bundling of deliverable files into a patch package and creation of informational material regarding the patches contents.

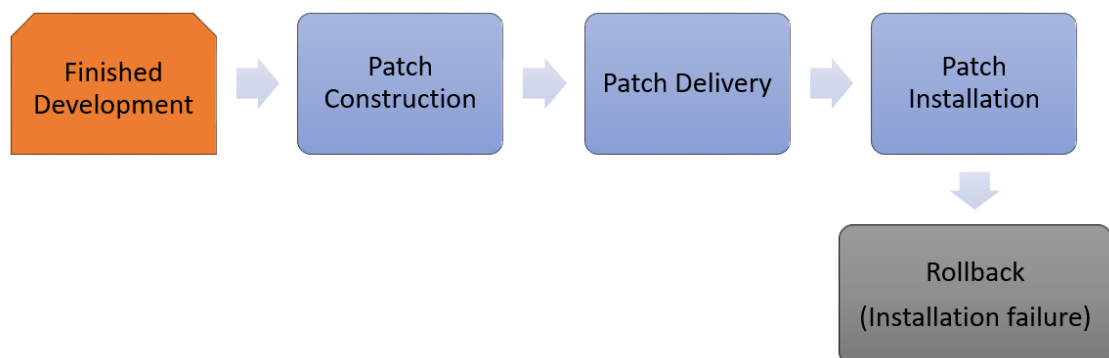
- **Patch Delivery:**

Sending of patch packages to customers which have self-hosted applications.

- **Patch Installation:**

Applying the patches contents to an application installation.

Figure 2: Patch Workflow:



Source: Own illustration

Once the process of finishing development is completed, patch construction begins. During patch construction, a delta analysis of the current application version and the previous version is conducted. This is performed to determine the necessary changes to the database and filesystem to turn an application from the previous version into the current one. Delta analysis consists of analyzing differences in the filesystem, e.g. files new to the target application version, files changed, files deleted as well as database table and table relation changes. Table changes may include any kind of table modification from a simple length change on a character field to the creation of a new table or alteration of primary keys in the target version of the application. Manual delta analysis is prone to error due to simple mistakes, e.g. not checking a folder of the installation and thus miss-

ing a changed file. Results from this operation are compiled into a user-friendly documentation of the necessary steps of installation as part of the release note. A complete patch package contains the following parts, further called artifacts of the patch package:

- SQL statements to update the database
- necessary files to overwrite or add
- list of files to remove
- descriptions of new features
- further descriptions of updated or altered functionality
- release note localized for the customers who should receive the patch. If the customer is in another country, the release note will most often be in English, otherwise “only” in the target language, for example German.

Compilation of this patch package is mainly done by a programmer of the software supplier as performing the delta analysis, and the creation of SQL statements is a technical task. Proofreading is done by testers, who also write the customer friendly descriptions of features and altered functionality. This is the general definition of a patch package. As an exception to the general definition of patch packages which are independent of the receiving client, there are also patches which may not be delivered to all customers. Those packages contain language specific translation packs for example. As a general rule, special patch packages may only contain changes of data, but no changes of structure, e.g. contents of a table may be modified, the table itself cannot. This is due to the fact that a change of structure would most likely break patch compatibility with further versions.

Artifacts of the patch construction part are then delivered bundled as said patch package via email to the administrators in charge of application installations. Those administrators are also in charge of storing the delivered patch packages and must keep track of installed patches and which patch version also called patch level is present on the test and the production environments. This part of the workflow is prone to error due to various reasons, for example, the receiving administrator's e-mail server may reject file attachments, and thus an alternative has to be used. Another example for a common problem is simple forgetfulness on the part of the distributor by forgetting to send the patch to an administrator. This can easily lead to problems installing future patches and – in a worst-case scenario – corrupt production data.

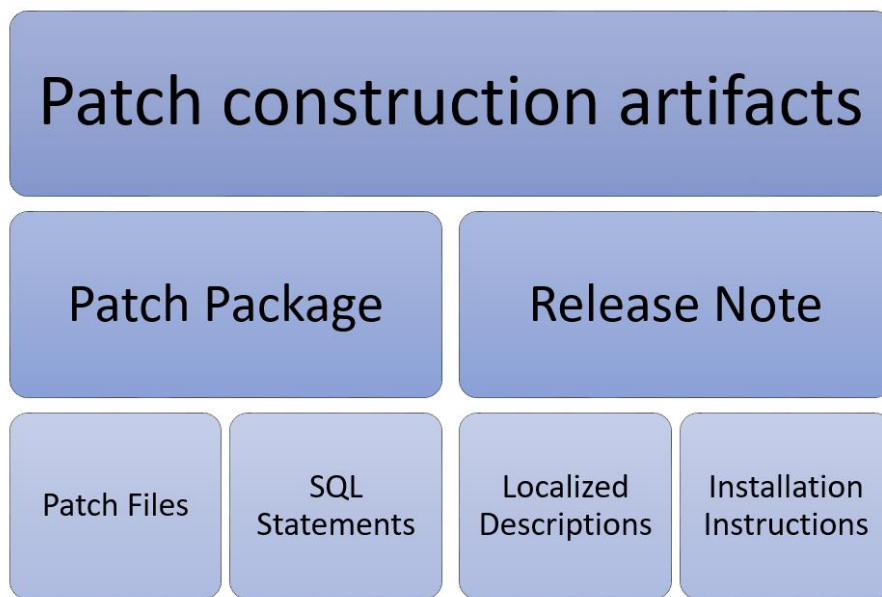
The Installation of patches is conducted by using the descriptions in the release note and is identical for all application installations and needs to be done in a sequential manner, e.g. from application version one to three; one cannot simply skip applying patch two.

The administrator in charge of an installation waits for approval of installation by the client and then proceeds per the release note. Problems may occur if the previous step failed and the administrators in charge tries to apply a patch while missing the previous one which often leads to an unstable application. Due to arbitrary patch levels on each customer's system and different versions in a single customer's environments, manual error correction measures conducted by the software supplier in case of a failed patch installation may prove difficult. In such a case the software supplier then needs to carry out another delta analysis of the client's system and the desired application version. Then the patch distributor needs to trace the steps performed per release note to determine the cause of the error. This process often proves to be problematic. On the one hand, the manual analysis is time-consuming on the other hand, there is no specific rollback procedure. A rollback procedure would describe how to create a partial backup of files and database parts, e.g. table rows, which are about to change. This way, a failed patch installation could simply be undone by applying the partial backup to restore the original files and data. To clean up the application of the partial backup, newly added files would be deleted and database changes reverted. Without a rollback procedure, however, the administrators mostly resort to a partial or full database and filesystem backup before installation of a patch. Partial database backups performed by the administrators are backups of entire tables instead of the whole database. In the case of a failed patch installation, this means that rollback of the entire database and filesystem takes as long as the manual error correction. This, however, depends on the size of the applications database in question, as some tables may contain gigabytes of data.

2.3 Problem and Goal Definition

The goal definition of this thesis is mainly derived from the previously explained workflow as this is to be automated completely, except the creation of release notes. Main problems which need to be solved or mitigated can also be derived from the workflows description and the artifacts created during patch construction:

Figure 3: Patch Workflow:



Source: Own illustration

The goals which need to be achieved are defined as follows:

Patch dependencies

The completed software solution needs to keep track of an existing patch package as well as which patch packages are required as dependencies to be applied before the patch itself can be implemented to the target installation. This mitigates the necessary delta analysis which needs to be conducted if a patch fails due to patch application mismatch by an administrator.

Patch library

A centralized solution is needed which stores constructed patch packages and their release notes in variously translated forms. This library is intended to replace a part of the patch delivery process as the actual packages do not need to be stored by administrators anymore. Newly created patch packages and their release notes need to be able to be

uploaded to the patch library. This lowers the number of faulty installations due to missing or forgotten patches, which have to be installed before the most recent patch may be installed. As the library would keep track of all patches and their dependencies and patch packages would be retrieved directly from the library.

Patch construction

An environment to create patch packages needs to be created which automatically performs the described delta analysis of the filesystem and the database and creates a patch package out of the information gathered. This patch package creation utility also needs to incorporate the previously mentioned dependency associations between patch packages. This way, all needed files will be included in the patch package, and all dependencies will already be taken into account.

Patch installation

An installation utility is required which automatically performs the necessary actions to apply the contents of a patch package to the software solution. This means it needs to update the filesystem and database. To mitigate the problem regarding rollback this utility also needs to be able to roll back to the previous application version and thus creating a partial backup of the application before performing the update process. Also, this utility needs to keep track of the installed version of the application and needs to be able to connect to the patch library to receive patch packages. The actual installation process needs to be able to be initiated by an administrator as soon as the patch package is ready to be delivered.

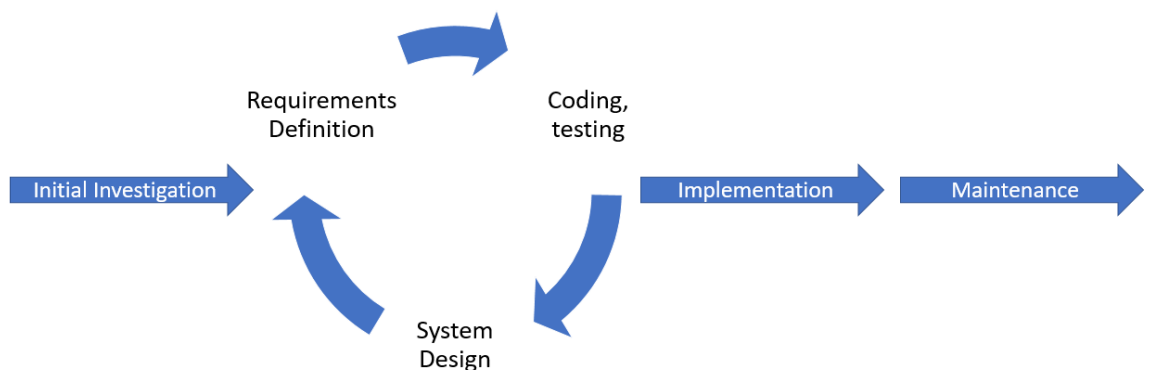
3 Elementary Concepts

This chapter describes the basic concepts used throughout this work. Concepts used are software development methodologies to match the previously as case study described process in a manner similar to abstract processes in literature. In literature, the whole concept processes regarding software are described as the software lifecycle; this describes the creation and maintenance of software as a whole.

3.1 Patch Deployment

Patch deployment can be described as the action of rolling out changes which need to be applied in order to lift a given program to a specific version. This needs to be done periodically during the standard software maintenance process. This maintenance process has different definitions or descriptions in software life cycle methodologies. For example, in the agile system development life cycle, patch deployment would be the crucial component to support “operate and support system in production”. In the agile or continuous delivery life cycles, it would be an element of “release solution”, as patch deployment is the actual release (installation) of changes.⁴ This concept is also vital in more traditional development methodologies like waterfall, prototyping or iterative development of software solutions as every development methodology needs maintenance or release component.⁵

Figure 4: Prototyping Software Life cycle:



Source: Own illustration based on CMS

⁴ See Ambler, Scott W. (2012).

⁵ See Office of Information Services (2008), pp. 1 – 3.

The software maintenance process is necessary because software solutions undergo changes over the life cycle and thus need to be modified while preserving the integrity of the software solution. There are different types of maintenance:

- Corrective changes
- Adaptive changes
- Perfective changes

Corrective maintenance may need to be conducted to mitigate errors in a software product so that it meets its requirements. These changes may need to be applied to an already deployed and actively used software solution as those errors may only be encountered in productive use. Adaptive as well as perfective changes are changes which enhance the software solution. Adaptive changes are necessary changes to accommodate a new environment, e.g. new functionality or changing of software requirements like the operating system. Adaptive changes are most likely required to guarantee function of a software solution when its basic requirements are updated, e.g. the underlying OS is updated, and an adaptive change must be made to ensure ongoing compatibility with the new OS version. Perfective changes do not alter existing functionality. Instead, they improve performance, maintainability or other characteristics of the software solution.⁶

⁶ See ISO14764 (1999), pp. 6 – 7.

3.2 Deployment Pipeline

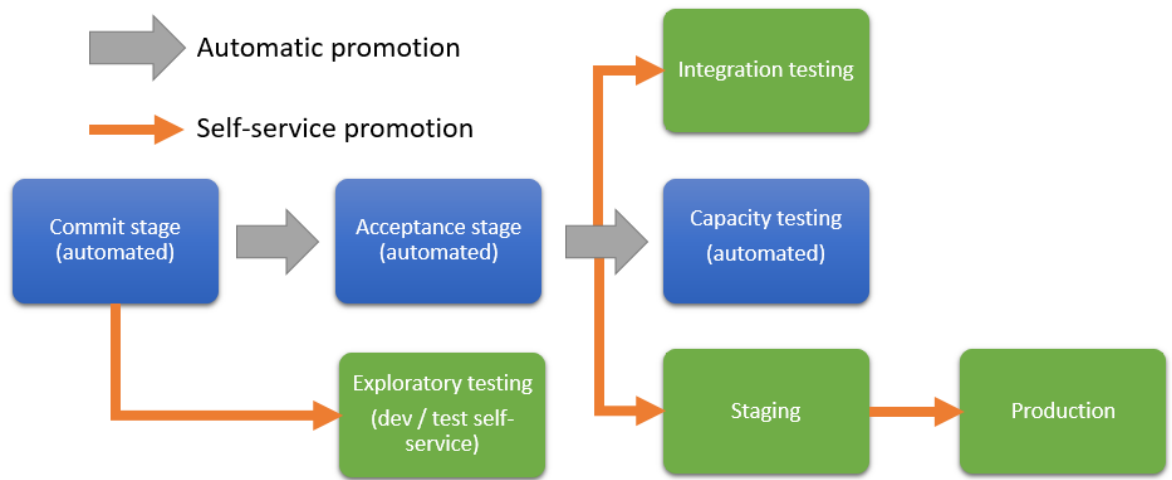
In general, a deployment pipeline can be defined as the automated process of necessary steps to assemble a fully functional software solution and deliver it to the software's users. The deployment pipeline starts with committing source code to a version control system and ends when the software is released to the users. The deployment pipeline involves building and testing through different stages until completion of the software.⁷

The term "deployment pipeline" emerged from projects at the company ThoughtWorks and can be considered a key pattern in continuous delivery of software solution designs. The deployment pipeline mainly involves automation to ease the process of deploying changes through different stages of environments, e.g. from development to test and then production. This automation principle was used to help ThoughtWorks overcome their struggle with complex and fragile deployments. A key goal of this was to create an entirely automated process to deliver deployments to any of ThoughtWorks environments in a fully scripted manner which does not need manual intervention to complete. To accommodate different environment settings configuration files would be used. The actual deployment pipeline pattern, described by Farley, starts upon committing changes to a version control system. The deployment pipeline then creates deployable packages and runs automated unit tests and other validations, like static code analysis against the newly created packages. If any of those checks fail, the pipeline stops at this stage. Any encountered errors must be corrected to proceed past this commit stage. Once all checks are completed the package is "promoted" to the next stage. In ThoughtWorks original pipeline the next stage consists of automated acceptance testing and afterward also automated capacity testing. Once it ran through all stages, it can be deployed on demand to any other stage of the systems environment for further manual tests, or it can be deployed to production.⁸

⁷ See Farley, David/ Humble, Jez (2010), pp. 106 – 107.

⁸ See Humble, Jez (2016), Continuous Delivery – Patterns.

Figure 5: Continuous Delivery Promotion:



Source: Own illustration reproduced from Humble, Jez (2016) Continuous Delivery

This promotion and stage concept heavily relies on the fact that packages are only built once and then recycled throughout the rest of the system landscape. It also implies two other paradigms which should be used according to Jez Humble: “Deploy the same way to every environment” and “Keep your environments similar.” Both paradigms basically describe a monoculture of systems as keeping them similar makes the deployment process recyclable.⁹

The deployment pipeline is also an integral part of the continuous delivery pattern, as this pattern aims for automation of all repetitive, error-prone activities in processes which are needed to deliver software.¹⁰ While a crucial part of continuous delivery, the deployment pipeline itself also serves additional purposes:

- Automation of build, test and deployment
- Visualizing the progress of software
- Finding and reducing bottlenecks in the deployment process

By providing these additional insights in the deployment and development process, the deployment pipeline acts as the realization part of a Value Stream Map, which is a key part of modern business practices.¹¹ Value Stream Mapping was developed in Supply

⁹ See Humble, Jez (2016), Continuous Delivery – Patterns.

¹⁰ See Skelton, Matthew/ O’Dell, Chris, (2016), p. 7.

¹¹ See Skelton, Matthew/ O’Dell, Chris, (2016), p. 27.

Chain Development for production management to identify and remove value waste inside companies.¹² While most modern business practices which target a business within the manufacturing industry are not directly applicable to software businesses, the common denominator of a Value Stream Map can help implementing them. This, in turn, makes it possible to profit from controlling mechanisms which were developed to help manufacturing companies.

As such the following benefits can be gained by using a build pipeline or a complete continuous delivery solution which implements one:

- **Accelerated Time to Market:** The release cycle is reduced drastically as new versions do not need to be excessively prepared. Average release cycles from conception to production has decreased from several months to two to five days.¹³
- **Improved Productivity and Efficiency:** Productivity and effectiveness are drastically enhanced by using a continuous delivery solution as it sets up test environments automatically. This task was previously a huge time sink for developers and testers alike.¹⁴
- **Reliable Releases:** The risks of rolling out a release are drastically decreased as the same scripts which control deployment onto test systems are used to deploy to the productive system afterward. Also, the size of releases is reduced as their frequency is increased. This results in fewer changes and thus smaller errors if they appear.¹⁵

For this thesis, however, the deployment pipeline starts from committing source code to a version control system and ends when the software is released to various server installations. This is due to the fact that this thesis considers an HTML5 client and PHP server application as a use case and the entirety of deployment pipelines is beyond this thesis scope. Thus, the actual release of client software to the applications users is handled through the PHP server backend. The deployment pipeline is only involved in preparing the software solutions client and not releasing it to the actual users.¹⁶

¹² See Hines, Peter/ Rich, Nick (1997), p. 46.

¹³ See Lianping, Chen/ Power, Paddy (2015), pp. 52 – 53.

¹⁴ Ibid.

¹⁵ Ibid.

¹⁶ See Chapter 2.1, figure 1.

3.3 Hashing

The process of hashing is used for generating a checksum to verify the integrity of a given set of data. For example, the most basic form of a checksum for a text file would be a combination of character counts:

This is a text-based example to demonstrate various checksums, starting from simple character counts and leading up to message digest techniques.

In this example text a character count based checksum would result in the following character table:

Table 1: Character Checksum Example

Character	e	s	t	a	i	o	c	r	n	m
Count	15	14	13	11	8	6	6	6	6	6

Source: Own illustration

To store this as a checksum, one could simply combine characters and counts like so: “a11c6e15i8m6n6o6r6s14t13”. Character count based checksums, however, do not guarantee that the given information is accurate or complete. They only ensure that the same number of symbols is present in each text with the same character counts. For instance, the previous example does not consider if a symbol is capitalized or not, the resulting checksum would be the same either way. To provide a better means to verify not only the correct amount of symbols but also verify the integrity, e.g. order of symbols and capitalization of a text file a message digest algorithm can be used. Message digest algorithms are based on the concept that a single bit alteration in the input message should result in a completely different checksum output.

The Secure Hash Algorithm, SHA, is designed for computation of a condensed representation of a set of binary data. It is based on the same principles as the MD4 message digest algorithm created by Ronald L. Rivest of MIT. The resulting message representation SHA generates is a 160-bit output which can be used to verify the contents of a set of data. The security of SHA is derived from the computational difficulty to find two different messages which result in the same message digest.¹⁷

¹⁷ See Jones, P. (2001), pp. 1 – 2.

SHA can currently be found in four different Versions (oldest to newest): SHA-0, SHA-1, SHA-2 (also known as SHA-256 among others) and SHA-3. To revert to the character checksum example, the following table compares the SHA-1 value of the sample text starting with an uppercase “T” and the SHA-1 value of the same text starting with a lowercase “t”:

Table 2: Character SHA-1 Example

Text	SHA-1 hash value				
Upper	7ad29cfc	0fc9ee1f	34274da3	3a488546	53781ac6
Lower	6f9b39d8	fe18ad3e	df40032a	c632d04c	5daf803b

Source: Own illustration

As it can be clearly seen in the above table both SHA-1 values, differ greatly as even a single bit difference results in completely different hash values by design of the algorithm. Even though SHA-1 is an improvement to its precursory algorithms like MD4 or MD5, it is by no means secure by today’s standards. Security in message digest algorithms can be best described by the difficulty to forge two distinct messages that result in the same message digest value; this is called a collision. Such collisions have been shown to be practically creatable with consumer grade GPUs for MD4 and MD5 algorithms.¹⁸

However, since 2005 SHA-1 was only vulnerable to a collision in theory. In late February of 2017 researchers from Google have proven in the field that SHA-1 is vulnerable to collision attacks by using Googles own computational resources and targeting weaknesses in the SHA-1 algorithm. They proved that it is possible to create colliding PDF documents on demand by using the equivalent computational power of 64 GPUs for ten days. This resulted in not only further deprecation of SHA-1 in the eyes of developers but also proved that it can be broken and hence must not be used in a future implementation and should be phased out of existing implementations.¹⁹

For actual usage and cryptographically secure means of obtaining a checksum to digital content, binary or otherwise, SHA-256 or similar algorithms should be used. In contrary to SHA-1s 160-bit message digest value, SHA-256 produces, as its name suggests, a 256-bit message digest value. Other algorithms, which are also approved by the Federal

¹⁸ See Intel (2013).

¹⁹ See Shattered.it (2017).

Information Processing Standard of the USA are SHA-224, SHA-256, SHA-384 and SHA-512 which all have message digest values corresponding to their naming scheme.²⁰

3.4 Compression

Transferring large amounts of data, can lead to many different problems. For example, it is possible that the target system only has a very limited amount of free storage capacity or that the mail rules state that no files larger than a certain size are allowed to be received. Furthermore, the bigger the file, the longer a transfer will take. This makes it essential to reduce the size of data without compromising the files themselves. This reduction of file size without compromising said files is called lossless compression. To reduce the size of data, a variety of algorithms can be applied. Some of those will be discussed below. Data compression can also be seen in an economical context as it trades traffic capacity versus CPU usage.

Deflate

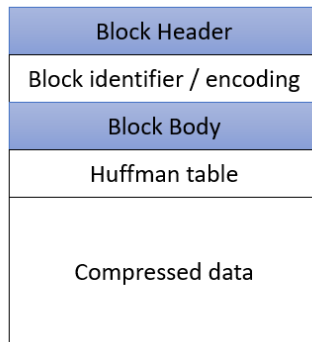
Most common compression techniques, such as ZIP, GZIP, and its derivatives, rely on the deflate algorithm.²¹ This algorithm basically reduces redundancy in a given data set while preserving all information and thus is considered a lossless data compression algorithm. Deflate can be seen as an algorithmic frame for compression steps as it does not actually compress data by itself.²² Instead, it uses different algorithms in a two-step process to compress data. Upon acting on a dataset which should be compressed the dataset will be split into blocks. Blocks are identified via a 3-bit header. This header contains all information on how to process the current block. For example, this information may declare that the next block is raw, uncompressed, data or a compressed block with a Huffman table supplied.

²⁰ See Hansen, T. (2011), p. 4.

²¹ See Deutsch, P. (1996b), p 1.

²² See Deutsch, P. (1996a), p 4.

Figure 6: Deflate compression block:



Source: Own illustration

Actual compression is also a two-stage process, the first step is “duplicate string elimination,” in this stage all duplicated strings in a given block are replaced with a placeholder that uses less space than the string it replaces. Algorithms used during this stage are LZ77 or LZ78. Both algorithms are also commonly known as LZ1 and LZ2. They are the basis for many variations of lossless data compression. Both create a dictionary of data strings, where the entries in said dictionary are the strings that have to be replaced. After creation of said dictionary, the algorithms replace occurrences of the dictionary entries in the dataset with references to the dictionary entry.²³ Once this processing of blocks is finished the “bit reduction” stage begins. In that compression stage, commonly used symbols in a given block will be replaced with shorter ones. This is done by using Huffman coding.²⁴

Huffman coding

This variation of encoding is used to reduce redundancy in a given set of binary data. Usually every binary word of a set of binary data is of the same length. Huffman coding describes the principle to reduce the amount of redundancy by assigning unique codes to symbols of said data.²⁵ This process results in a binary tree, the so-called Huffman tree. By weighting a Huffman tree, which means using the smallest codes for the most frequent encountered symbols it is possible to compress data. Consider the following example:

TEST TEXT FOR TESTING PURPOSES

²³ See Lempel, Abraham/ Ziv, Jacob (1977).

²⁴ See Deutsch, P. (1996a), p 1 – 2.

²⁵ See Huffman, David A (1952) p. 1098 – 1100.

Symbols of this text are characters and can be encoded as follows (most occurrences, results in smallest symbol length):

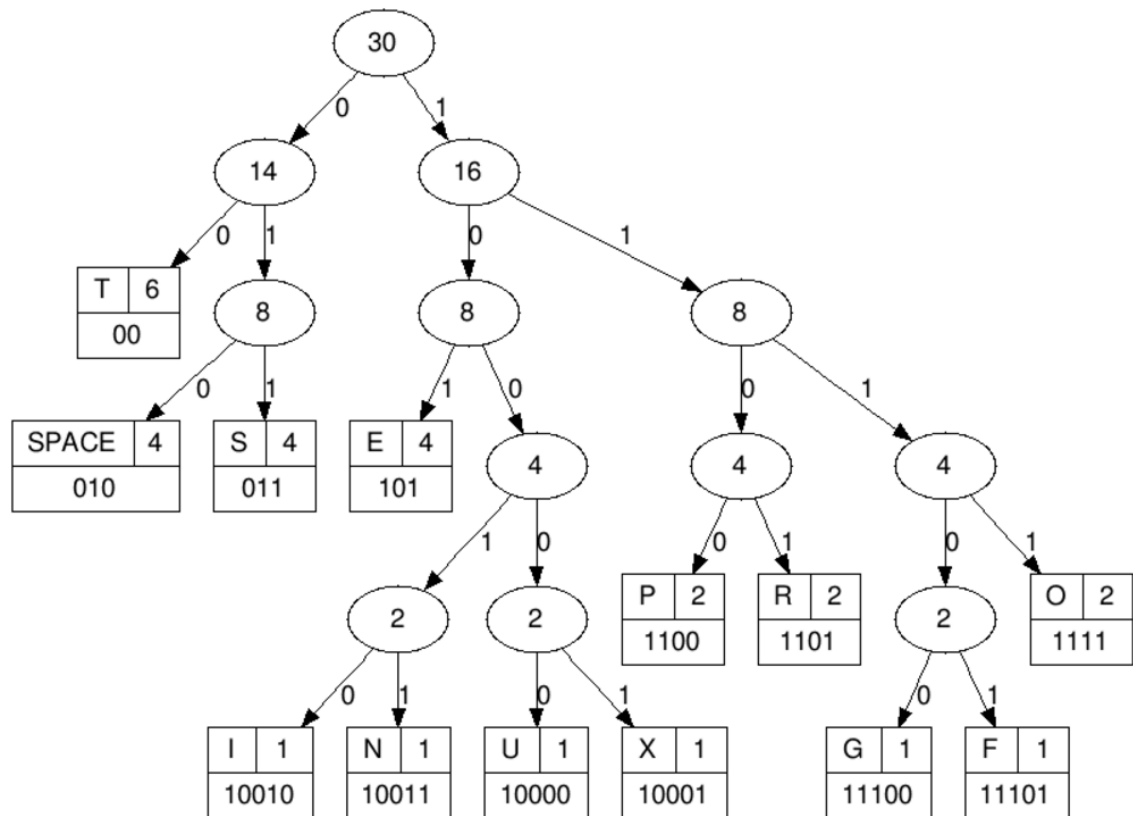
Table 3: Huffman coding symbols

Symbol	Occurrences	Code
T	6	00
SPACE	4	010
S	4	011
E	4	101
P	2	1100
R	2	1101
O	2	1111
I	1	10010
N	1	10011
U	1	10000
X	1	10001
G	1	11100
F	1	11101

Source: Own illustration

This table of codes can be generated by creating a weighted binary tree first. To create said tree the symbol count is examined first, in the above example there are 30 characters in total. Out of those 30 characters, two groups can be formed; one contains 14 characters which contain the most frequently used ones, the other 16 characters are the least used. The character "T" is most frequent and thus assigned the symbol "00", afterward each branch is examined, and the most common character in each group is assigned a symbol. This step is then performed again until no characters are left to allocate symbols to:

Figure 7: Huffman tree:



Source: Own illustration made with Huffman Tree Generator²⁶

Assignment of symbols within groups of the same occurrence does not matter as there is no reduction to gain. It only matters that the more common ones are assigned the smallest possible symbol.

²⁶ For further information, see Huffman Tree Generator.

4 Evaluation of existing solutions

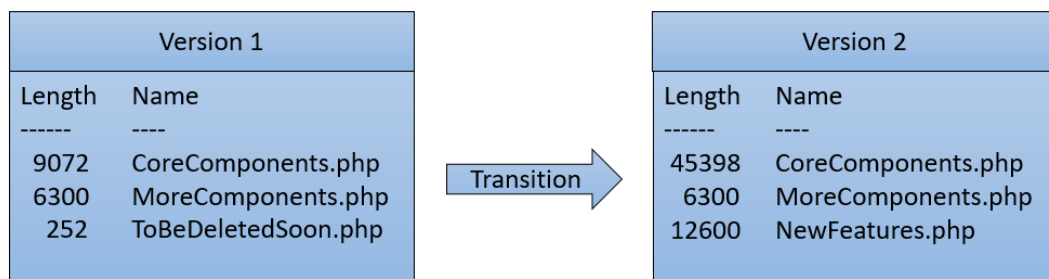
The following chapter will introduce existing solutions to partial problems outlined in the case study. The two main topics discussed will be approaches to synchronize the filesystem and the synchronization of the database in the context of the application described in the case study. In this thesis, synchronization describes the process of transforming a given, versioned, filesystem structure or database to a new version. Basis for this transformation process and the following descriptions of file synchronization processes is a simplified form of the synchronization definition that was also used for the Unison file synchronizer specification.²⁷ The simplified form used in this thesis assumes that propagation of changes is one-way and overrides any local changes.

4.1 Filesystem

Filesystem synchronization, in its simplified form, is mostly comparable to creating versions via some form of source control system as both solve the same task of keeping files and folder structures in order. This is due to having one “master” filesystem which simply overrides any filesystems which need synchronization, analogous to a “master” repository in source control systems.

For example, consider the following filesystem layouts. The left filesystem will be called “version one” and the right filesystem “version two”:

Figure 8: Basic filesystem version example:



Source: Own illustration

²⁷ See Pierce, Benjamin C./ Vouillon, Jérôme (2004), pp 1 – 4.

See also Balasubramaniam, S/ Pierce, Benjamin C. (1998), pp 2 – 4.

To synchronize the “version one” filesystem layout to a “version two” filesystem layout, the following steps need to be taken into account:

- The file “ToBeDeletedSoon.php” needs to be removed.
- The file “NewFeatures.php” needs to be added.
- The file “CoreComponents.php” has changed its file size and thus needs to be updated as its contents seem to have changed.
- The third file “MoreComponents.php” does not appear to have changed as its file size is the same. Hence, this file can remain untouched if its contents do not have changed.

This synchronization process gets more complicated as more files are involved. For example, in a standard file hierarchy, there can be files with the same name in different locations at the same time.²⁸ Those files do not necessary have the same contents, nor the same meaning as a unique file is defined per its filename in combination with its location:

Figure 9: Advanced filesystem version example:

Version 1						Version 2		
Length	Directory	Name				Length	Directory	Name
3441		index.php				3441		index.php
7472	API	accounts.php				7472	API	accounts.php
7472	API	basicLogin.php				2642	API	main.php
1337	API	main.php				7825	Server	database.php
7825	Server	database.php				5491	Server	main.php
5491	Server	main.php				9984	Server	modules.php
9984	Server	modules.php				4992	GUI	client.js
4992	GUI	client.js				1024	GUI	extensions.js
624	GUI	frame.html				624	GUI	frame.html

Transition

Source: Own illustration

Most of the changes which have to be conducted seem analogous to their counterpart from the basic filesystem example. However, the main distinction is that the file “main.php” is existent in the API subfolder as well as the Server subfolder. This leads to the conclusion that the filename itself is not a unique identifier.

To reduce ambiguity files should be referred to by their relative path within a given folder. In this example, they would be referred to as “API/main.php” and “Server/main.php”. To

²⁸ See Balasubramaniam, S/ Pierce, Benjamin C. (1998), p 6.

perform a synchronization of the advanced “version one” filesystem layout to its “version two” filesystem layout, the following steps need to be taken into account:

- The file “API/basicLogin.php” needs to be removed from the API subfolder.
- The file “API/main.php” needs to be updated.
- The file “GUI/extensions.js” needs to be added to the GUI subfolder.

This process of file synchronization does not take into account that a file could have changed its contents while its size remained the same.

4.2 Git

Git is a version control solution initially created by Linus Torvalds and is widely used worldwide. The most active provider of repository space, GitHub, currently hosts 52 million repositories and has 19 million active users.²⁹ These statistics only include the public or private hosted repositories in GitHub's system. They do not include self-hosted repositories, local repositories or other hosting providers.

Git was created to build a free version control system that can support the development of the Linux kernel. Up to that point, the free version of BitKeeper VCS (version control system) had been used to develop the Linux kernel. However, new restrictions on the free version of BitKeeper VCS made it unusable for Linux. Linus had to search for alternatives. Thus, began the creation of Git. It was meant to offer features, which were not present in previous free version control systems, namely the following requirements should be met:³⁰

Facilitate distributed development

Git should enable parallel, independent and simultaneous development in different repositories without the need to constantly synchronize with a central repository. This should allow multiple developers in multiple locations to work on the same project, even if some were temporarily offline.³¹

Scale to handle thousands of developers

Because Git was primarily created to facilitate the development of the Linux kernel, simple distributed development was not the only requirement. The Git system should enable an enormous amount of contribution from an unknown number of developers each Linux release. Thus, Git should provide means to integrate those contributions reliably into one release.³² This requirement stems from the fact that for each Linux kernel release roughly 800 to 1.100 contributors and their contributions have to be handled reliably.³³

²⁹ See GitHub (2017).

³⁰ See Loeliger, Jon (2009), pp. 2 – 3.

³¹ Ibid.

³² Ibid.

³³ See Loeliger, Jon (2009), p. 217.

Perform quickly and efficiently

Another requirement to handle an enormous amount of contributions while keeping the version control system stable and reliant, Git should ensure fast network transactions and a relatively small memory footprint. To achieve these goals, compression and “delta” techniques would be needed to reduce the amount of stored and transferred data.³⁴ The “delta” techniques are basically techniques to acquire the delta of two files to alter one file into the other.

Maintain integrity and trust

To enforce integrity and confidence in a distributed version control system and to ensure that no files have been altered in the transition from one repository to another Git uses the SHA-1 cryptographic hash algorithm.³⁵

Enforce accountability / Immutability

As another fundamental aspect of a distributed version control system, Git keeps track of who changed which files and, if possible why they were changed. This is the reason why Git enforces a change log on every commit of changes into a repository. While the contents of the change log are left to the developer who committed the change, enforcing a change log creates an accountability trail for all changes. This, in combination with Git’s paradigm of immutability, creates a version control history in which every change to every file can be traced to its cause. Git’s repository database is per definition immutable, which means that once a file has been placed inside, it cannot be removed without creating a trail of changes. Those files also cannot be altered in any way, as deleting a file and storing the same file again also results in a log of changes.³⁶

Atomic transactions

At the basis of Git’s repositories are transactional commits, those commits may involve one or more files and are tracked as a single changeset. Per Git’s design all repository updates are atomic, which means that either the entire change set of a given commit is applied to a repository or no change at all. This eliminates the possibility to update a repository into a partially updated and hence corrupted state.³⁷

³⁴ See Loeliger, Jon (2009), pp. 2 – 3.

³⁵ Ibid.

³⁶ See Loeliger, Jon (2009), pp. 2 – 4.

³⁷ Ibid.

Support and encourage branched development / Complete repositories

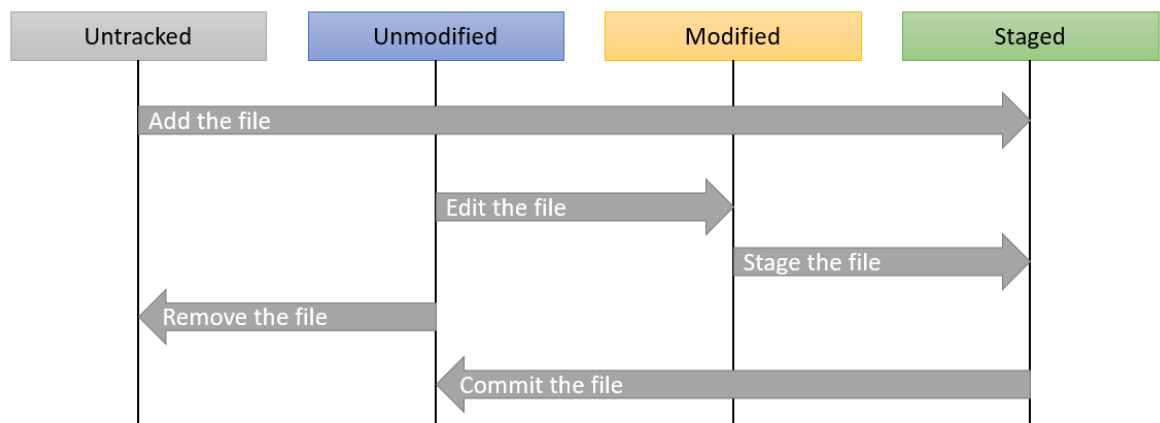
Branched development is yet another requirement stemming from the Linux kernel era of Git's creation as new features would be developed in a version of the repository which branches off from the main development branch. This feature is especially valuable to support distributed repositories where no central master repository exists. To support this feature, one of Git's essentials is a clean and fast merging of commits which would conflict upon application. This feature goes along with "complete repositories," as no single repository can be a central one which would be queried for historical revisions, each cloned repository contains all historical revisions for every file and thus is basically a standalone repository.³⁸

A clean internal design

Git facilitates an object model with simple structures to capture fundamental concepts for raw data, directory structures, changes and a globally unique identifier model which was created with a distributed repository management in mind.³⁹

Git allows staging changes before they are committed to a repository. Staging is meant to be used to prepare a commit to help to do clean commits. To accommodate various file changes the following principle to record variations in a Git controlled folder is used:

Figure 10: Git file Status:



Source: Own illustration reproduced from Git⁴⁰

³⁸ See Loeliger, Jon (2009), pp. 4.

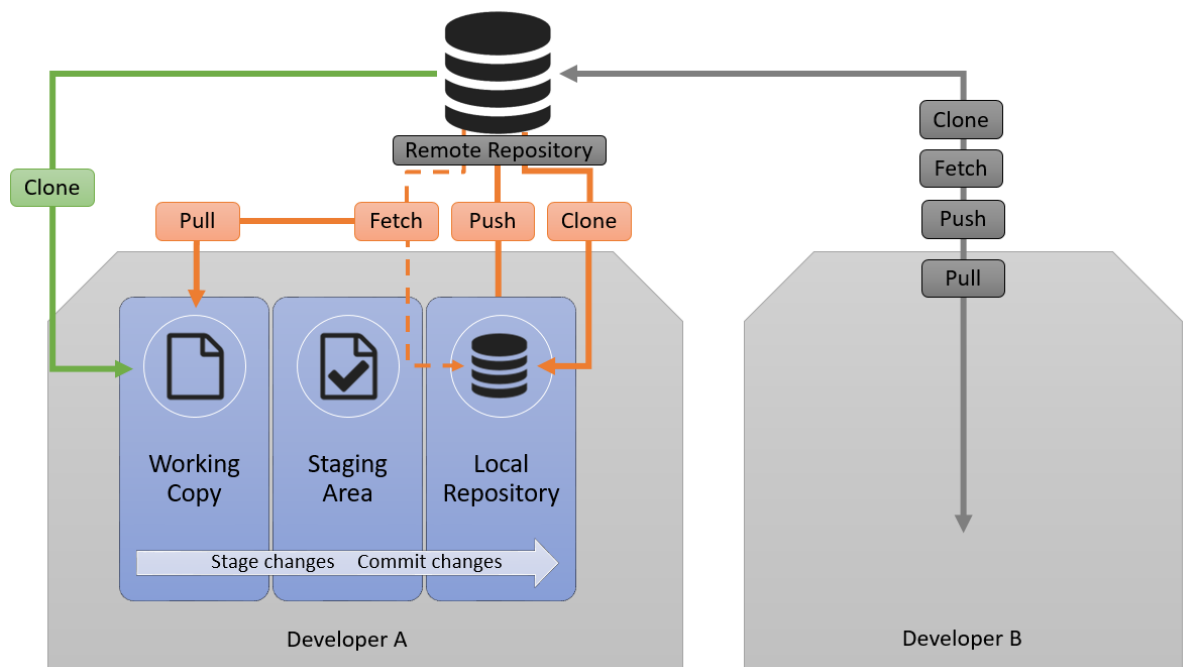
³⁹ Ibid.

⁴⁰ See Git – Basics.

Files in a folder are either tracked or untracked; untracked means that this file has been removed. Modifying a file leads to Git flagging a file as modified. If changes are staged or committed, all files return to the status unmodified. The staging step can also be skipped. This way staging and committing would be treated as if they were the same.

To facilitate development on a project in a shared repository, a developer's local repository can be synchronized with a single remote repository from which the other developers update their own local repositories:

Figure 11: Git remote repository:



Source: Own illustration reproduced from Git-tower⁴¹

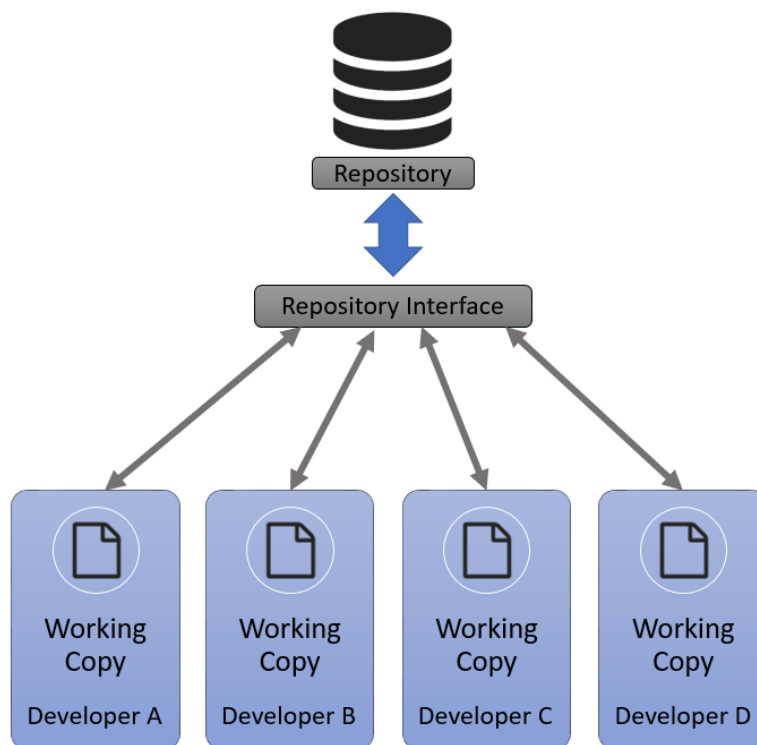
The remote repository can simply be cloned to a developer's machine; this, however, does not simply create a file and folder structure copy of the remote repository. Instead, it creates an entire copy of the repository, including the commit history. This local repository is in all things equal to the remote one. Upon altering files of this local repository, no other developer has access, even if those changes are committed. As staging and committing takes place on the local repository. To make changes to the remote repository a commit from the local repository can be pushed to it. In return to receive changes which were made to the remote repository commits can be pulled into the local repository.

⁴¹ See Git-Tower (2017).

4.3 Subversion

Apache Subversion (commonly only referred to as Subversion) is a version control solution maintained by the Apache Software Foundation and aims to be a universally recognized and adopted open-source software to facilitate centralized version control of files.⁴² Subversion was developed as a successor to the Concurrent Versions System (CVS), which was the main version control solution in the open source world. As CVS development became more difficult due to fixing flaws with the software the originators of Subversion took it upon themselves to create a new software solution which attempts to avoid CVS flaws.⁴³

Figure 12: Subversion repository:



Source: Own illustration based on Subversion's architecture⁴⁴

The Subversion system was designed with a single repository in mind, as such the Subversion server is the only entity holding a complete repository. The repository works like a typical file server, clients can connect to it and write or read files. In contrast to a file

⁴² See Subversion (2016).

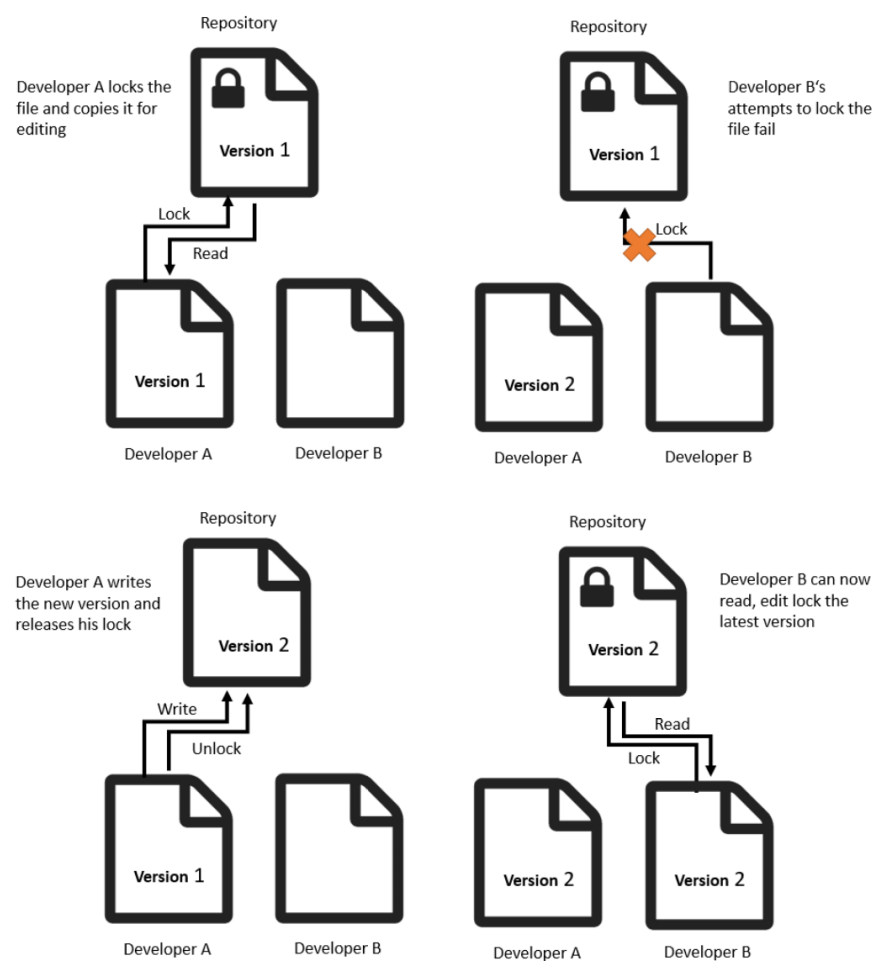
⁴³ See Collins-Sussman, Ben/ Fitzpatrick, Brian W./ Pilato, C. Michael (2011), xiii.

⁴⁴ See Collins-Sussman, Ben/ Fitzpatrick, Brian W./ Pilato, C. Michael (2011), xvi.

server, however, the repository logs all changes which have been made and thus it is possible for a client to request an arbitrary revision of a file. Every client of the Subversion server may request to have a local working copy of the repository. This essentially means that one revision of the repository is copied over to the client as a local file and folder structure. The working copy is intended as a complete local version to be worked upon, e.g. a complete PHP project to be used and altered locally. After modifying the working copy, changes which were made locally would be written to the repository by Subversion's client.⁴⁵

Subversion supports two different modes to deal with concurrent working repositories. As previously mentioned a single Subversion server may let any number of clients connect and in turn every client can create a working copy and modify said copy. To handle this problem, one way supported by Subversion is to let clients lock files:

Figure 13: Subversion locks:



⁴⁵ See Collins-Sussman, Ben/ Fitzpatrick, Brian W./ Pilato, C. Michael (2011), p. 2.

Source: Own illustration modified from Subversion⁴⁶

Locking a file prevents anyone else, except the client, who created the lock to change the file or lock it themselves. In the previous figure “Developer A” got a lock for file “Version 1” to create file version two. “Developer B” is thus unable to get a lock on “Version 2” until the lock is released. As “Developer A” writes his changes to the repository and releases the lock “Developer B” is able to retrieve “Version 2” and lock the file. This procedure, however, leads to an increased overhead in coordination as a locked file needs to be unlocked either by the user holding the lock or a Subversion repository administrator. If for example, Developer A forgets to release his lock and is on vacation, Developer B must contact the Subversion administrator to release the lock.⁴⁷

As an alternative to locking files to facilitate the parallel development of a repository Subversion also offers the “copy-modify-merge” model. In this model, every user may change his working copy as he wishes. Upon requesting to write the changes back to the repository, it may happen that another user recently updated a file that was now modified in the working copy. If this happens, the writing user receives a “file is out of date” error as it would override changes which were already written. To resolve this problem changes which were written to the repository can be merged into the working copy. This is meant to be used to address potential conflicts and then write the resolved version back to the repository. The “copy-modify-merge” models benefits are that, while working concurrently, no user of the repository has to wait for writes and locks.⁴⁸

⁴⁶ See Collins-Sussman, Ben/ Fitzpatrick, Brian W./ Pilato, C. Michael (2011), xvi.

⁴⁷ See Collins-Sussman, Ben/ Fitzpatrick, Brian W./ Pilato, C. Michael (2011), p. 4.

⁴⁸ See Collins-Sussman, Ben/ Fitzpatrick, Brian W./ Pilato, C. Michael (2011), p. 6.

4.4 Database

Synchronization of the filesystem was relatively straightforward; synchronization of the database involves a more complex process. To illustrate the difficulty of synchronizing the database consider the following example of a simple database table:

Figure 14: Basic database version example:

Version 1

Column Name	Datatype	PK	NN
id	VARCHAR(40)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
ip_address	VARCHAR(16)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
user_agent	VARCHAR(50)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
last_activity	INT(10)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
activity	TEXT	<input type="checkbox"/>	<input checked="" type="checkbox"/>
users_id	INT(10)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
creation	DATETIME	<input type="checkbox"/>	<input type="checkbox"/>

Version 2

Column Name	Datatype	PK	NN
id	VARCHAR(32)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
access	INT(10)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
data	TEXT	<input type="checkbox"/>	<input checked="" type="checkbox"/>
username	VARCHAR(60)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
user_agent	VARCHAR(80)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
user_agent_name	VARCHAR(30)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
user_agent_version	VARCHAR(30)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
platform	VARCHAR(30)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
ip_address	VARCHAR(45)	<input type="checkbox"/>	<input checked="" type="checkbox"/>
referrer	VARCHAR(128)	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Source: Own illustration

In this example, there are some changes which need to be performed to turn a version one table into a version two table. In a simplified form, the needed changes are as follows:

Table 4: Performed changes to example database

Type	Action
Change	“id” from 40 to 32 characters
Change	“ip_address” from 16 to 45 characters

Change	"user_agent" from 50 to 80 characters
Remove	"creation", "last_activity", "users_id"
Rename	"activity" to "access"
Add	"username", "user_agent_name", "user_agent_version", "platform", "referrer"

Source: Own illustration

Keep in mind, that this is a simplified form because while transitioning from version one to version two the primary id column "id" changed its length. This is a special case because it is a unique identifier and reduction of length leads to a loss of information which needs to be handled. Precautions need to be taken to treat this information loss if the "id" column is referenced by another column. Other changes which introduce a loss of information is the removal of columns, these are taken as necessary information loss as version two does not use them anymore.

Changing a database table is not only a change of structure. Newly added columns may need to be initialized or flagged, and relations to other tables may need to be created, altered or dropped. As seen in Figure 5, all newly created columns are flagged as NotNull (NN) for example. These changes also need to be taken into consideration when a roll-back solution is created. To revert those changes, loss of information, which occurs by removing columns, needs to be taken special care of. This information loss cannot be reversed by the MySQL database system.

4.5 Database Migration Tools

Most database migration tools are specialized solutions which are mostly created by a software company to resolve their specialized version of the problem to migrate databases from one version to another. As such there are no complete software products available. The tools can be mostly seen as script libraries, and there is an enormous amount of scripts written by lone programmers to ease database migration. To use a suitable base to discuss database migration techniques and tools only those maintained by a company will be further reviewed and considered in this thesis. The libraries which will be addressed in this chapter are from SoundCloud and Facebook.

SoundCloud – Large Hadron Migrator

SoundCloud's Large Hadron Migrator (LHM) library for database migrations was developed with Ruby on Rails in mind and to mitigate the specific problem of ALTER TABLE statements on tables which contain millions of records. As altering a table of such size leads to downtimes of an hour or more. Developers tend to design around the problem by expanding the database instead of altering it and using join tables. This, however only leads to the same problem. Adding or changing indices to optimize data access becomes just as difficult if the database grows even further. The LHM is able to perform migrations online without locking the table while the system is live. Due to the fact that LHM is written in Ruby for Ruby on Rails, it needs an active ActiveRecord connection. LHM also presumes a single numerical primary key which is auto-incremented as per Ruby on Rails convention.⁴⁹

Ruby on Rails limits the use of primary keys for tables to a single auto-incremented integer column called id. As this is the standard in active record migrations and active record convention definition of a primary key, there is no support for composite primary keys.⁵⁰

⁵¹ This, in turn, leads to the conclusion that LHM does not natively support composite primary keys either.

⁴⁹ See SoundCloud LHM (2016).

⁵⁰ See Ruby on Rails – AR Migrations.

⁵¹ See Ruby on Rails – AR.

Facebook – Online Schema Change for MySQL

Facebook’s Online Schema Change (OSC) is a library similar to SoundCloud’s LHM and was developed for a similar use case, namely to perform ALTER TABLE changes without a downtime and in a fast and reliable manner. OSC has been built with PHP and is available as a PHP library. The basic procedure which is performed by OSC to make changes to a table is as follows:

Table 5: OSC Table Change Phases Overview

Step	Description
Copy	The table structure which is to be modified is copied.
Build	Schema changes are made to the copy and data is loaded into it.
Replay	Changes which were made to the original table are copied over to the copy.
Cut-over	The copy table is renamed to the original, and thus both tables are switched, another replay may be needed.

Source: Own illustration

During the copy phase, the contents of the table which is to about to be modified are copied in batches of rows into files. This is done to avoid heavy load on the MySQL server as using “Insert into X select * from Y” would produce. Afterward, two copies of the table structure are created, one is named deltas and is used to keep track of changes made to the original while the copy is being modified and loaded with data. This is done by setting up an update trigger on the original table which keeps tracks of changes and thus inserts those changes into the deltas table. After creation of the deltas table, the previously generated files are used to populate the copy table which is now modified. Afterward, during the replay phase, changes which were made to the deltas table are replayed to the copy table. To finish the modify operation, during the cut-over phase, the original and the copy tables get swapped. A cleanup is needed to remove newly created tables like the deltas table.⁵² OSC has no special requirements regarding naming of columns or usage of primary keys as the primary key is accessed by its index name. This leads to a compatibility with composite primary keys.⁵³

⁵² See Callaghan, Mark (2010).

⁵³ See Facebook OSC, line 827 for further information about handling of primary keys.

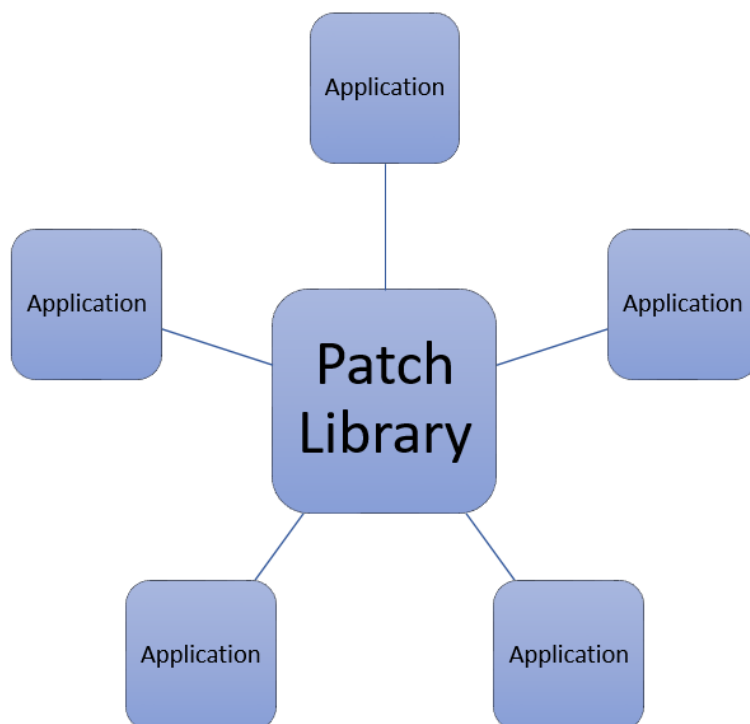
5 Conception

Based on the research from the previous chapters, the following chapter of this thesis will incorporate the explained concepts and existing solutions. Beginning with the platform choice for development, followed by the creation of a general process model based on the case study (see 2.2) and the goal definition of this thesis. The following describes how the evaluated solutions can be integrated into a process model. After a recap of the integration potential of the evaluated solutions, the chapter ends with the definition of a data model for prototyping.

5.1 Architecture and Platform Choice

To make a platform choice for development the case study's goals must be taken into consideration. One goal is to provide a centralized patch library which controls the distribution of patch packages and release notes on demand:

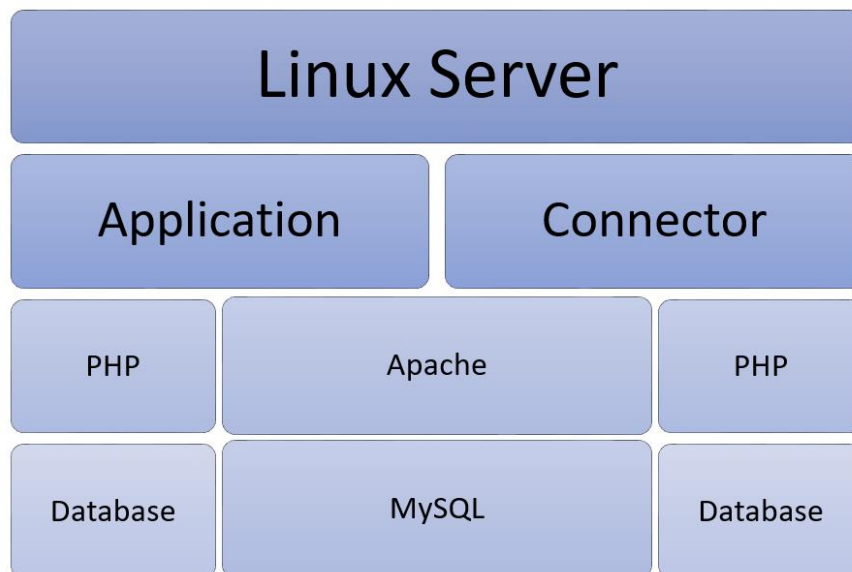
Figure 15: Patch Library:



Source: Own illustration

This, in turn, means that all instances of application backends need to be connected to the central patch library on demand if an action is requested which needs data from said library, like installing patches. To make such connections possible, without integrating a connection solution into every application supported by the patch library, a separate utility application is needed. Separating the patching utility from the application which it needs to patch also has the benefit of separation of concerns. In the case of a patch installation failure, the patch utility application would still be responsive. This is due to the fact that the source code of the patch utility would not be intertwined with the patched applications that are then in an unstable state. As such the solution to develop must consist of two components. One component is the patch library which creates, stores and delivers patches. The other component is a utility which is installed on the web server to facilitate automated deployment by connecting to the patch library and perform the patch installation. This utility is further called “Connector”. To also provide an opportunity to utilize resources already present on the target server which hosts the application, already installed software environments can be used:

Figure 16: Patch Connector:



Source: Own illustration

The Connector can be written in PHP to utilize the already installed Apache web server. As the Connector needs access to the database of the application to perform database changes, which may be required by patch packages, it also needs access to the database of the application. As the Connector already needs a MySQL database connection to perform this task, it can also use another database on the same MySQL server to store version information required for its function. Data necessary for its function are for

example credentials needed to authenticate against the patch library and data about the currently installed version of the application.

5.2 Process Modelling

Since the case study already divided the whole workflow of patch deployment into three distinct processes, those processes will now be used as the basis of process modeling in regard to the previously mentioned structure of different components of the software solution which is to be created.⁵⁴

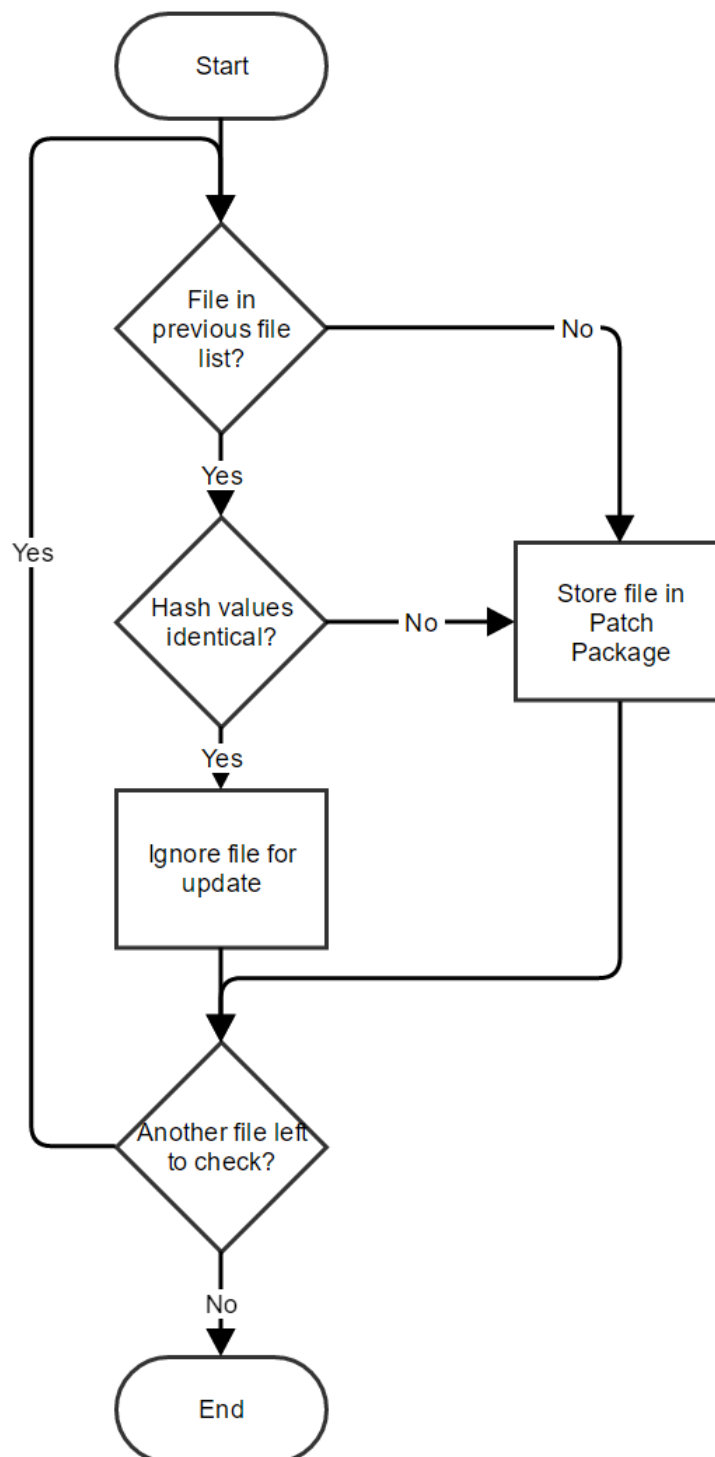
The defined processes can be further broken down as follows:

Step 1: Patch construction

The actual construction of a patch consists of two distinct artifacts, namely the patch package and its corresponding release note. Defined goal for this process is that the delta analysis is performed automatically and resulting artifacts are stored in the patch library. The patch library also needs to know which version was newly created to handle dependencies correctly. Conducting delta analysis of the filesystem can be performed by firstly creating two lists of files. One file list contains every unique file identifier and the hash value of the previous application version and the second file list contains every unique file identifier and the hash value of the target application version. By iterating through the entire file list of the current application version and completing the following check process for each file, the files in need of adding to the application or update can be identified:

⁵⁴ See Chapter 2.2.

Figure 17: Filesystem Delta Analysis:

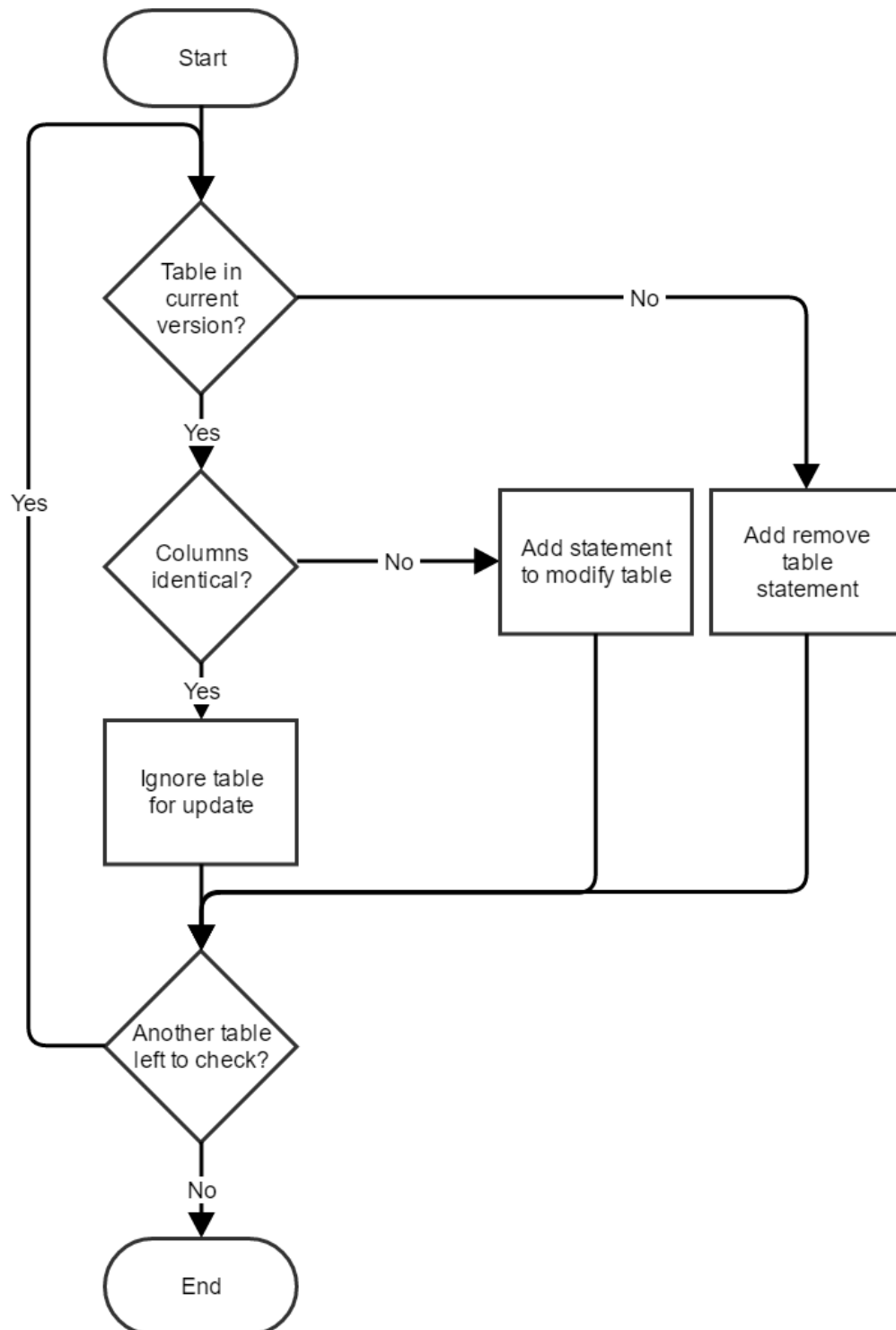


Source: Own illustration

Afterward, the current file list needs to be matched against the previous one again to determine files which need to be removed upon patching the application; those should be added to a list of files to be deleted and added to the patch package. As the entire file

is missing, there is no reason to match hash values or other specialized checks. After completion of the file system delta analysis, the following delta analysis needs to be conducted for the database by iterating through the tables as if they were files:

Figure 18: Database Delta Analysis:



Source: Own illustration

Each table of the last application version needs to be checked, if it is not in the current version, the table needs to be removed. If it is, the structure of the table needs to be checked to determine if it was modified. If a modification is found, the changes need to be replicated via a SQL statement that is added to the patch package. If a new table was found which is in the current version, but not in the last application installation, the table needs to be created.

Step 2: Patch delivery

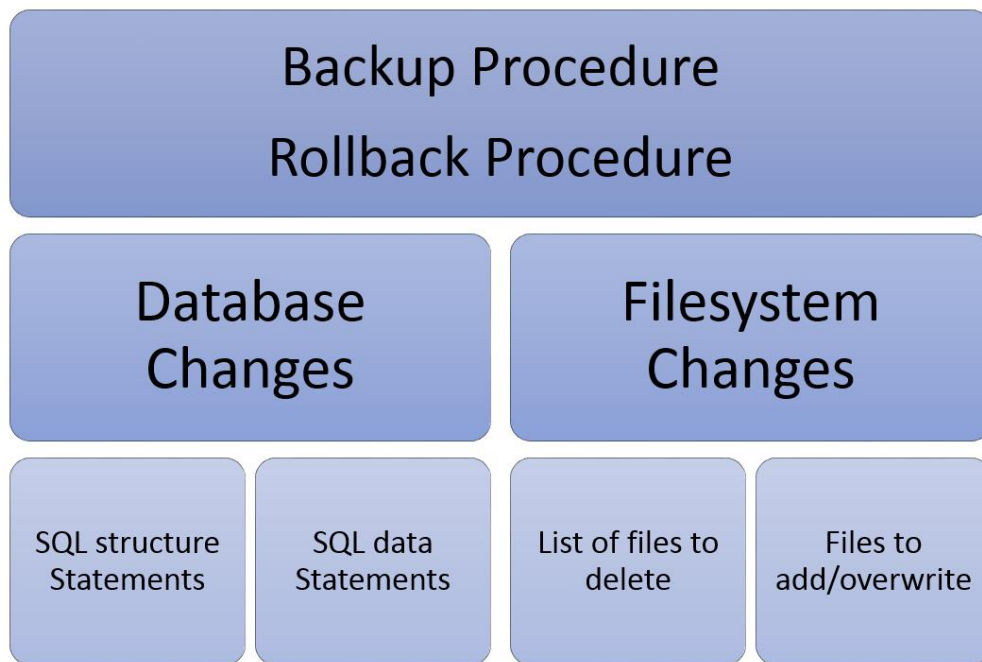
Patch delivery can be automated through the Connector application. As the Connector application is able to make a connection to the patch library at any point in time, there is no need to send the patch packages and release notes to the administrators beforehand. Therefore the Connector should provide a GUI which can show the current application version and its applicable release note. The Connector should also be able to list all available patches and show their dependencies. An administrator can then retrieve patches and their related release notes on demand through the Connector's GUI. The Connector also stores the installed application version, and thus there is no further need to keep track of installed patches by the administrator. In conclusion, previously mentioned patch delivery problems are entirely mitigated by fulfilling these goals of patch delivery:

- Unreliable transmission
- Human error during delivery (recipients forgotten)
- Human error during storage (wrong order of patch)

Step 3: Patch installation

Patch installation is conducted by the Connector application on demand. If the administrator accesses the interface of the Connector, the currently active patch version is shown as well as the patches which can be applied. Upon request of installation, the patch package is downloaded from the library. The patch package needs to include the following information, which in turn needs to be extracted by the Connector:

Figure 19: Patch Package Contents:



Source: Own illustration

Installation:

Steps for applying the Patch Packages contents during the installation procedure are as follows:

- Backup: the backup procedure needs to be performed, on this stage all files which are about to change need to be stored in a backup location. For the database backup, all tables which are about to change have to be dumped into files.⁵⁵ In case of drastic alterations of tables, for example changes to primary keys or similar, specific statements need to be included which revert the database to a state that can use the backup table.
- Database: afterward the database changes can be applied by executing the extracted SQL statements.
- Filesystem: the filesystem changes are performed by first copying the files to add or override into the application's source directory. Afterwards, cleanup is done by deleting files according to the list provided by the patch package.

⁵⁵ Reference to chapter 4.5 OSC for MySQL load benefits.

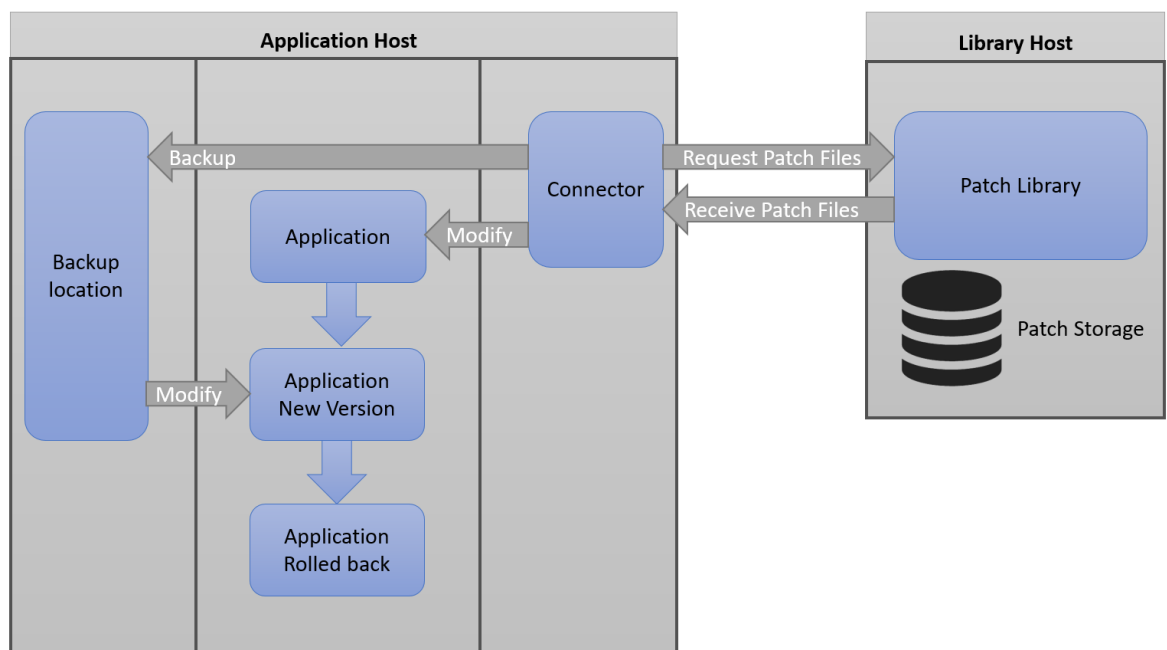
Rollback:

If there are any problems during installation, a rollback procedure needs to be triggered. This rollback procedure will be performed as follows:

- **Filesystem:** rollback can be carried out by simply reversing the application of filesystem changes: namely deleting all files which were included in the list of files to add or overwrite. Afterward, files which were deleted or overwritten by application of the patch can be restored by copying over the old files from the backup location.
- **Database:** changes can be reversed by switching the database backup tables with the changed ones. This, however, would mean that SQL structure statements do not contain primary key definitions, alterations or similar. In that case, a specific statement to revert those changes would be needed as rollback procedure.

To further illustrate these processes, consider the following figure:

Figure 20: Patch Installation Workflow:



Source: Own illustration

This diagram shows the interactions between the four distinct locations; backup, application, Connector and library host. Only the library host is remote, other locations are on the application host's system. After requesting the patch package (patch files) other tasks are done locally through the Connector application. It creates and controls the handling

of backups and thus is able to roll back the application without further requests to the library.

5.3 Integration of Solutions

The following chapter describes which solutions will be integrated into the prototyping phase of this thesis. It will also show how the solutions and techniques explained in previous chapters can be integrated.

Compression

Compression will be utilized to form single-file patch packages out of all necessary files which are required by a given patch. This makes handling of patch packages easier as a complete patch can be stored as a single entity. Network connections which transfer the patch package also just need to transfer a single set of binary data. It also allows to reduce the file size of a given patch package by utilizing compression. Due to the complete patch package being a single file, compression also helps in providing the means of obtaining a checksum, as a single set of data is required to obtain one.⁵⁶

Hashing

To verify integrity of a patch package, the SHA-256 Secure Hashing Algorithm will be used to assign every patch package its own hash value. This will be used to ensure that the file transfer was successful and that the file has not been altered in transit.⁵⁷ To utilize this functionality, the patch library holds all hash values to its stored patch packages and transfers the hash value to a requesting Connector before sending the patch package. After the hash value has been transmitted, another transfer connection will be opened to obtain the patch package. After receiving the patch package, the Connector is then able to compute the packages hash independently and check both hashes. Performing this step with two connections has the benefit to ensure that the checksum is transferred by a different connection, which, in turn means, that a potential file corruption would need to affect both connections in order to remain undetected.

Version Control System

Implementation of a Version Control System (VCS) will be utilized if it enables a shortcut in the implementation of file system synchronization. Both previously discussed systems

⁵⁶ See chapter 3.3.

⁵⁷ Ibid.

(Git and Subversion) have drawbacks regarding usage as file synchronization tools when compared to the defined algorithmic steps to patch an application file system. Drawbacks of Git for practical usage would be, for example, that the entire repository needs to be copied into every installation of the application. This also leads to an installation of a Git client in every Connector application, as otherwise handling of entire Git repositories would require construction of a system which could handle Git repositories. However, even if implementation of VCS systems for file synchronization does not prove to be practical, the patch system should at least support the necessary means to create patches from Git controlled application installations, as Git could be used in its development. This support would at least ignore Git specific files upon delta analysis as they have no meaning for the usage of the application and thus do not need to be included in pack packages.

Database Migration Tools

From the mentioned Database Migration Tools, LHM and OSC, the library made by Facebook will be used as possible.⁵⁸ This is due to the fact, that LHM is based on the Ruby on Rails Database schema and thus a Connector application would need to provide a Ruby on Rails application environment. In addition to the dependency to Ruby, LHM is constructed to utilize Ruby's Active Record solution, which means that it does not support composite primary keys. The application which needs to be patched however may require a composite primary key, as it is not built upon such an Active Record solution. Facebook's OSC also has incompatibilities regarding its usage. As OSC is primarily built with MySQL version 5.1 in mind. It also checks for this version in its source code and cannot be used if a connection is made to a newer database server. OSC is also a quite an old library as it was published in 2010 and not recently updated, as such there is no guarantee regarding its usage. However, it still can be utilized to form the basis to perform the actual database modification through the Connector. OSC has been constructed to ease the alteration of MySQL tables while retaining the integrity of the database, as such, it creates partial database backups and delta tables which can be utilized to fulfil this goal. These goals are already implemented in OSC with PHP and MySQL in mind and as such can be salvaged to create the Connectors database patch deployment.

⁵⁸ See chapter 4.5.

5.4 Data model

To provide a holistic handling of automated patch installation, it is necessary to expand the defined artifacts created during patch construction by adding information about the steps of applying the patch. This includes installation, backup as well as the rollback steps necessary in a structured and unambiguous manner which can be utilized by a program. The previously employed patch creation workflow included most of this information in the release note, compiled to be understood by administrators performing the installation manually.⁵⁹ For use by a program, a data structure containing the following would be required:

- Structure format describing file changes
- Structure format describing database changes
- Structure format describing rollback procedure if specified

To store the required information, a data format needs to be used which can be utilized by both; the patch library and the Connector application. If possible, the format should be readable by a human and thus should not be binary. For such a task, there are two widely adopted data exchange formats: XML and JSON. Extensible Markup Language (XML) is a tag-based text format to store data. Separating content with tags gives them meaning per the used XML structure⁶⁰:

Figure 21: Patch Structure XML Example:

XML Patch Container
<pre><?xml version="1.1" encoding="UTF-8" standalone="yes"?> <patch> <version>1.1</version> <files> <add>NewFile.php</add> <delete>SoonToBeRemoved.php</delete> </files> <database> <structure>NewStructures.sql</structure> <data>NewDefaultSettings.sql</data> </database> </patch></pre>

Source: Own illustration

⁵⁹ See chapter 2.2 Patch construction artifacts.

⁶⁰ See W3C – XML.

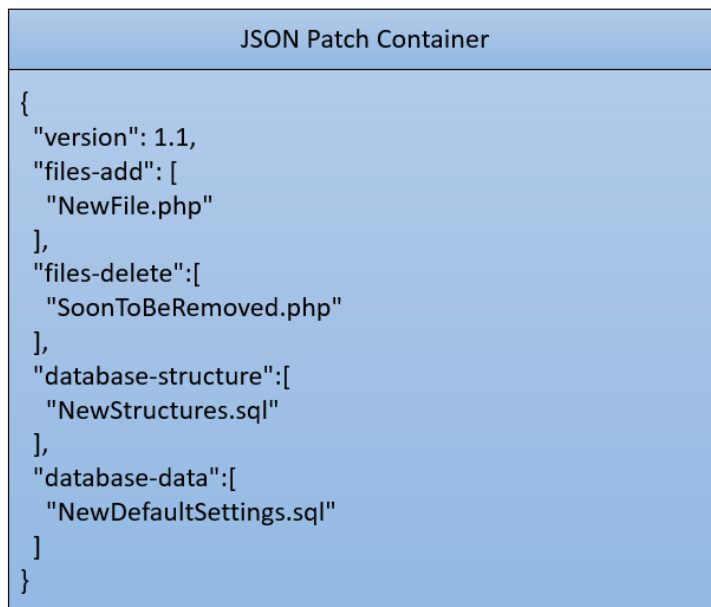
As seen in the figure above, all information that had been declared as needed, would be stored. However, the XML data format is quite verbose as it requires to store information in a tag enclosed manner. The “Patch Structure XML Example” basically contains the following information:

- Add “NewFile.php”
- Delete “SoonToBeRemoved.php”
- Modify database structure by executing “NewStructures.sql”
- Add data by executing “NewDefaultSettings.sql”

As can be clearly seen in the example and its associated instructions above, XML used in this, basic, form is already verbose.

The JavaScript Object Notation (JSON) data format is not as verbose as XML though it was created with similar goals in mind. JSON was defined to provide a lightweight, text-based, language-independent data interchange format which was derived from the ECMAScript Programming Language Standard.⁶¹ To revert to the previous example, in JSON it would look like this:

Figure 22: Patch Structure JSON Example:



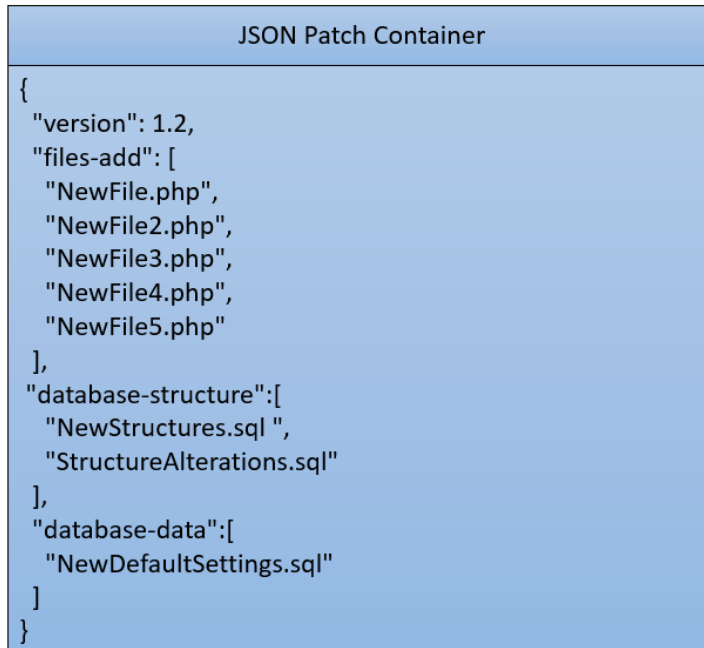
```
{
  "version": 1.1,
  "files-add": [
    "NewFile.php"
  ],
  "files-delete": [
    "SoonToBeRemoved.php"
  ],
  "database-structure": [
    "NewStructures.sql"
  ],
  "database-data": [
    "NewDefaultSettings.sql"
  ]
}
```

Source: Own illustration

⁶¹ See Crockford, D. (2006), p. 1.

At first glance the JSON example looks quite similar regarding verbosity. However, due to a format structure without tags, new files can simply be added without the need to provide a new open and closing tag for each element. Consider the following example:

Figure 23: Patch Structure JSON Example Extended:



```
{
  "version": 1.2,
  "files-add": [
    "NewFile.php",
    "NewFile2.php",
    "NewFile3.php",
    "NewFile4.php",
    "NewFile5.php"
  ],
  "database-structure": [
    "NewStructures.sql ",
    "StructureAlterations.sql"
  ],
  "database-data": [
    "NewDefaultSettings.sql"
  ]
}
```

Source: Own illustration

In this example, a lot of file related changes were added, while the JSON structure remained relatively similar and less cluttered. In contrast, consider the necessary structure for the files in XML:

Figure 24: Patch Structure XML Example Files:

XML Patch Container
<pre> <?xml version="1.1" encoding="UTF-8" standalone="yes"?> <patch> <version>1.1</version> <files> <add>NewFile.php</add> <add>NewFile2.php</add> <add>NewFile3.php</add> <add>NewFile4.php</add> <add>NewFile5.php</add> </files> <database> <structure>NewStructures.sql</structure> <structure>StructureAlterations.sql</structure> <data>NewDefaultSettings.sql</data> </database> </patch> </pre>

Source: Own illustration

The XML version is cluttered with tags whereas the JSON version uses less space to describe the same data. Adding more files to the JSON version only increases the array declaration "files-add":["...", "..."] slightly while adding new files to the equivalent XML duplicates its tag-based information for every file.

JSON and XML are both integrated in the PHP runtime as core components⁶², however while JSON is integral part of PHP, compilation of the PHP runtime can be done without the XML library⁶³. Due to this possibility of deactivated XML and the complicated access to XML parsing (there are many libraries for XML⁶⁴), JSON will be used. JSON can also easily accommodate a variety of different information without complicated changes.

⁶² See PHP – JSON.

⁶³ See PHP – XML.

⁶⁴ For further information see SimpleXML, DOM and XML parser in PHP

6 Prototyping

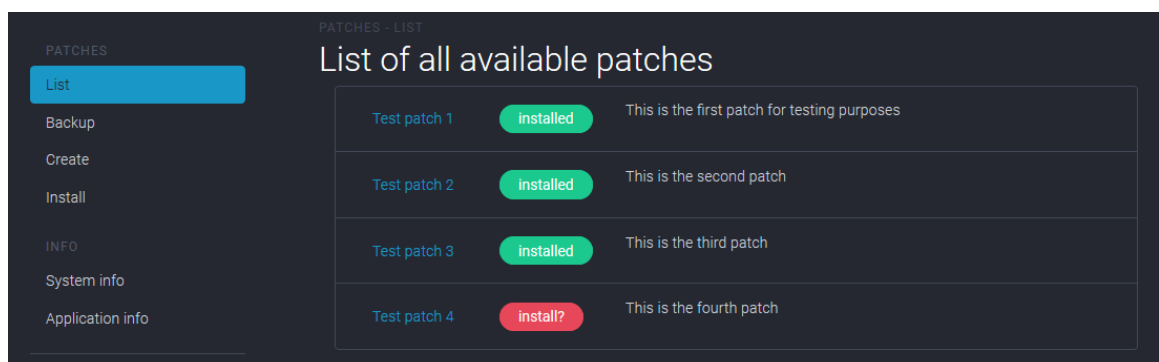
The prototype solution is built as two standalone PHP applications which communicate via HTTP. The basic structure for the Connector applications is as follows:

- HTML5 frontend, built with AngularJS to provide a GUI
- PHP endpoints to expose patch functions to frontend
- Local MySQL database to store information about patch controlled application

The basic interactivity between these parts is as follows: the GUI can be requested by accessing the Connectors base path via a browser, e.g. for a local XAMPP developer environment, typing "localhost/connector" in the browser's URL field. Endpoints contain the backend functions which need to be executed to perform patch related tasks. All endpoints include the same basic PHP file for initialization which establishes a connection to the Connectors own database and the applications database which it controls. Every access to the endpoint functions is done through the HTML5 GUI via HTTP calls.

Upon accessing the Connector's HTML5 GUI, a list of all available patches and their installation state is shown:

Figure 25: Connector Client - GUI:



Source: Own illustration

The GUI is divided into two parts. On the left side is the navigation, other parts of the Connector's functionality are available via links. On the right side is the content area where applicable information and form data is displayed for the selected functionality.

The before mentioned patch list retrieves the available patch listing from the library application and enriches it with locally stored information about the installed patches. In the library's database, the following table stores all installed patches:

Table 6: Patch Table

Column	Type	Description
id	integer	Unique id of the patch package
name	varchar	Display name of the patch package
version	varchar	Version identifier
info	text	JSON object containing patch metadata
data	longblob	Binary column containing patch archive data

Source: Own illustration

The “patches” table is used by the patch library to store the entirety of available patches. This makes compatibility between both systems and comparison of patches a non-issue as the information contained is only needed by the Connector during an installation procedure and transferred on demand.

The Connector also stores additional information in a “system” table, which contains “key – value” information:

Table 7: Connector Application State

Name	Value
installed_structure	JSON object containing the entire file and database structure of the current application version
installed_versions	JSON object containing a list of all installed patches
library_host	URL of the Patch Library to connect to
version	Application version identifier

Source: Own illustration

Data contained in this table is used to identify the installed application version quickly and to hold the entirety of the last application structure in a JSON dataset. This is used later to perform delta analysis. The list of installed versions contains the ids of every applied patch to display the patch list and enrich it with the information about installation state.

6.1 Delta Analysis

Delta analysis can be started from the “Create” section of the Connector. The “Create” section offers two buttons “Start Delta Analysis” and “Create Patch Package”, the “Start Delta Analysis” function must be used first and is built as follows:

Figure 26: Connector Frontend - getDelta:

```
$scope.getDelta = function() {
    $http.post('library/endpoints/create.php', {type: 'delta'})
        .then(function(response) {
            $scope.response      = response.data;
            $scope.show_response = true;
            $scope.show_result   = false;
            $scope.show_additional = false;
        });
};
```

Source: Own illustration

Upon clicking the button “Start Delta Analysis”, a simple HTTP call is made from the HTML5 frontend to the Connector’s “create” endpoint. This requests processing according to the requested type “delta”. The “delta” action from the “creation” endpoint extracts the stored information from “installed_structure” first to acquire a starting point for needed delta checks. Then it begins collecting a flat file list of the entire application folder:

Figure 27: Connector Backend – File Listing:

```
//retrieve current listing and sha256 hashes
$list_new = array();
foreach (glob_recursive($app_path.'*') as $filename) {
    //ignore directories from listing
    if(!is_dir($filename)){
        $list_new[] = array(
            'file' => str_replace($app_path, '', $filename),
            'hash' => hash('sha256', file_get_contents($filename))
        );
    }
}
```

Source: Own illustration

Collecting said file list is done by recursively iterating through the entire application directory and storing two values per file: the unique file identifier with a reduced, relative, path and the calculated sha256 hash of the files contents. The reduced path is due to

platform independent storage, file functions in PHP use an absolute path. Further processing needs a relative path, thus the absolute path to the applications directory is removed. Directories are not added to the list because they are indirectly stored in every unique file identifier, if a file is within a directory. For example: “directoryname/file-name.fileextension” implicitly contains the directory “directoryname”, thus the information that “directoryname” exists does not need to be determined or stored. Empty directories are omitted, as they add no value.

Afterwards, there are two arrays:

- \$list_new, which contains the entirety of the current application file structure.
- \$list_old, which contains the complete file structure of the last version.

Now, enough information is gathered to proceed according to the defined algorithm for file system delta analysis. As for comparison purposes, two complete lists are required.⁶⁵ Determination of “added” files that means files which have changed or have been added and “deleted” files are done as follows:

Figure 28: Connector Backend – File Delta:

```
//retrieve changed / added files
$list_changes = array();
foreach($list_new as $file){
    if(!in_array($file, $list_old)){
        $list_changes['modified'][] = $file;
    }
}

//retrieve list of deleted files
foreach($list_old as $file){
    if(!in_array($file, $list_new)){
        $list_changes['removed'][] = $file;
    }
}
```

Source: Own illustration

These loops represent the defined algorithm in chapter 5.2 in a simplistic form. This is due to the implicit comparison operation of the “in_array” function. Basically “in_array” checks if the supplied “needle” entry exists in a supplied “haystack”, which is an array. This function iterates through the array and checks with an explicit type equality operation if a matching entry was found. This also enables using an array in place of the “needle” to check an array of arrays for the occurrence of an array with exactly the same

⁶⁵ See chapter 5.2 Patch Construction.

values. Due to this behavior of the “in_array” function and the previous storing of sha256 hashes in file lists, a simple check if a file entry of the \$list_new array is also in \$list_old is enough to determine all changes and added files. Determination of removed files is basically the reverse operation, checking every entry of \$list_old and determine if a file is not in \$list_new. This results in an array containing changes \$list_changes[‘changes’] and an array containing removed files \$list_changes[‘removed’].

Afterward, delta analysis for the database needs to be performed. To apply the algorithm defined in chapter 5.2, the database structure needs to be obtained in a format which can be acted upon in PHP. This is done by creating another multi-level array structure, similar to the file lists for the current version \$database_new:

Figure 29: Connector Backend – Database Listing:

```
//retrieve current database tables
$STH = $DBH_application->prepare("SHOW TABLES");
$STH->execute();
$STH->setFetchMode(PDO::FETCH_NUM);
$database = array();
while($row = $STH->fetch()){
    $database[] = $row[0];
}

//retrieve current columns list
$database_new = array();
foreach($database as $table){
    $database_new[$table] = array();
    $STH = $DBH_application->prepare("SHOW FULL COLUMNS FROM `{$table}`");
    $STH->execute();
    $STH->setFetchMode(PDO::FETCH_NUM);
    while($row = $STH->fetch()){
        $database_new[$table][] = array(
            'name'      => $row[0],
            'type'      => $row[1],
            'collation' => $row[2],
            'null'      => $row[3],
            'key'       => $row[4],
            'default'   => $row[5],
            'extra'     => $row[6]
        );
    }
}
```

Source: Own illustration

By using the database connection to the applications database and using MySQL’s “SHOW TABLE” command, a list of all tables is created. The “SHOW TABLE” command only lists non-temporary tables.⁶⁶ This table list is then enriched by information about the

⁶⁶ See MySQL – SHOW TABLES.

columns, the table is made out of by using “SHOW FULL COLUMNS FROM `<table-name>`”. This command lists technical details about a table’s columns. These details are needed to determine which changes have been made; the following particulars are returned:

Table 8: MySQL “SHOW FULL COLUMNS” Result:

Return value	Description
Field	Name identifier of the column
Type	MySQL data type of the column
Collation	Collation of character like columns
Null	Specifies if Null is an acceptable data value for the column
Key	Specifies if the column is a key and if so, which type of key. E.g. primary, unique
Default	Default value of the column. If a value had been omitted during insertion, this value would be used
Extra	Additional information, like auto_increment or on update functions tied to the column
Privileges	Displays which privileges are set on the specific column
Comment	Displays the database comment for the column

Source: Own illustration based on MySQL⁶⁷

For the sake of simplification, information about needed privileges and the database comment are discarded and not used in further comparison.

The array \$database_new now contains all structural information about every table in the applications database and thus is the database equivalent of \$list_new. Its counterpart, \$database_old is retrieved from the Connector’s system tables “application_structure” entry and has the same format:

⁶⁷ See MySQL – SHOW COLUMNS.

Figure 30: Connector Database Structure Example:

```

"database": {
  "example_table": [
    {
      "name": "id",
      "type": "int(11)",
      "collation": "",
      "null": "NO",
      "key": "PRI",
      "default": "",
      "extra": "auto_increment"
    },
    {
      "name": "char_field",
      "type": "varchar(30)",
      "collation": "latin1_swedish_ci",
      "null": "NO",
      "key": "",
      "default": "",
      "extra": ""
    }
  ]
}

```

Source: Own illustration.

The database structure contains all necessary information to produce an alter table or create table statement. The actual delta check is as follows:

Figure 31: Connector Backend – Database Delta:

```

//retrieve added / changed tables and columns
$databases_changes['modified'] = array();
foreach($databases_new as $table_name => $table){

    //new table detected
    if(!isset($databases_old[$table_name])){
        $databases_changes['modified'][$table_name] = $table;
        //has the table structure changed?
    }else if($databases_old[$table_name] != $table){

        $databases_changes['modified'][$table_name] = array();
        //compare table fields
        foreach($table as $field => $column){
            if(!array_key_exists($field, $databases_old[$table_name])
            || $databases_old[$table_name][$field] != $column){
                //add changed column to changes array
                $databases_changes['modified'][$table_name][] = $column;
            }
        }
    }
}

```

Source: Own illustration

By iterating through the current applications database structure, all table entries are checked against their counterpart from the old version. This is done in a similar way as the previous check for files, but instead of using “in_array” the equals operation is performed on both entries in question directly.

If a change is detected, that means a table is not exactly like it was before and checks need to be done for every column of that table. If such a check detects a variation from \$database_old the entry in question is added to \$database_changes['modified'], which has the same structure as the complete database structure descriptions but only holds changed columns. Afterward, the removed tables and columns need to be identified, this is done like so:

Figure 32: Connector Backend – Delta Removed:

```
//retrieve removed tables and columns
$database_changes['removed'] = array();
foreach($database_old as $table_name => $table) {
    //removed table detected
    if(!isset($database_new[$table_name])){
        $database_changes['removed'][$table_name] = $table;
        //has the table structure changed?
    }else if($database_new[$table_name] != $table){

        $database_changes['removed'][$table_name] = array();
        //compare table fields to find removed columns
        foreach($table as $field => $column){
            if(!array_key_exists($field, $database_new[$table_name])
                || $database_new[$table_name][$field] != $column){
                //add removed column to removed array
                $database_changes['removed'][$table_name][] = $column;
            }
        }
    }
}
```

Source: Own illustration

First all table names are checked against the \$database_old array to determine if a whole table was removed. If this is true, the table in question is added to \$database_changes['removed']. If the table exists in the new structure, another check is performed to determine if the table equals the old version. If this check fails, every column of the current version is compared to the new version to determine if a single column has been removed.

Lastly, the result of the delta analysis is sent back to the requesting frontend:

Figure 33: Connector Backend – Delta Result:

```
echo json_encode(
    array(
        'files'           => $list_changes,
        'files_complete'  => $list_new,
        'database'        => $database_changes,
        'database_complete' => $database_new
    )
);
```

Source: Own illustration

The sent data includes change sets for database and file structure and current, complete structures for the filesystem and database.

6.2 Create Patch Package

After the frontend receives the delta analysis results, it enables the “Create Patch Package” button which is tied to the “store” function. It also shows the complete result of the previous delta analysis operation for inspection of correctness by the user. On the first click, the “store” function displays a form which needs to be filled to proceed creating a patch package:

Table 9: Connector Create Patch Form

Field	Type	Description
Patch name	Textfield	Name to display in patch list
Optional	Checkbox	Determines if patch increases patch level
Description	Textarea	Text to display in patch list
SQL to modify database	Textarea	SQL to upgrade to the current version
SQL to revert database	Textarea	SQL to revert changes made by the modify SQL
Release note	Upload	File upload for release note pdf files

Source: Own illustration

The information which has to be provided by the user, is information which cannot be automatically collected during the delta analysis stage or be created in the next stage.

Input such as the patch name and description for display in the patch list and the decision whether or not the patch is optional needs to be entered by the user who creates the patch. The release note needs to be created externally as it is a single PDF file which needs to be uploaded. As an automatic generation of SQL statements by using the detected change sets is beyond the scope of this thesis, the user is prompted to enter a set of SQL statements which would modify the database and another set which would undo the changes if applied. To ease this process, the result of the database delta analysis is shown next to those text areas like so:

Figure 34: Connector Frontend – Show Database Changes:

```
{
  "modified": {
    "example_table": [
      {
        "name": "example_column",
        "type": "varchar(75)",
        "collation": "latin1_swedish_ci",
        "null": "NO",
        "key": "",
        "default": null,
        "extra": ""
      }
    ]
  }
}
```

Source: Own illustration, for a full table example, see appendix 1.

This is necessary information to find the tables in question and to create SQL statements to alter them accordingly.

After entering the additional required information, activating the “store” function again sends the change sets and additional data to the Connector’s backend to create a patch.

Figure 35: Connector Frontend – Store:

```
$http.post('library/endpoints/create.php', {type: 'store',
                                             data: $scope.response,
                                             form: $scope.model
}) .then(function(response) {
    $scope.result = response.data;
    $scope.show_result = true;
    $scope.show_response = false;
    $scope.show_additional = false;

    //clear locally stored patch info
    $scope.response = '';
});
```

Source: Own illustration

The Connector then proceeds with building the actual patch package. Locally stored information about the delta analysis is cleared once the response of the “create” endpoint is retrieved. After evaluation of the supplied data on the backend, the Connector proceeds by creating a temporary ZIP archive file:

Figure 36: Connector Backend – Create Patch Package: Files

```
if($zip->open($zip_file, ZIPARCHIVE::CREATE) === true){
    //add modified files if supplied
    if($files_to_store){
        foreach($files_to_store as $files){
            $zip->addFile($app_path.'\\'.$files['file'], 'files\\'.$files['file']);
        }
    }
    //add list of modified files to archive
    if($files_to_store){
        $temporary_file_modified = sys_get_temp_dir().'\\modified_files';
        file_put_contents($temporary_file_modified, json_encode($files_to_store, JSON_PRETTY_PRINT));
        $zip->addFile($temporary_file_modified, 'modified_files.json');
    }
}
```

Source: Own illustration

Collecting all files needed to update the application is performed by iterating through the `$files_to_store` array which was previously the `$list_changes['modified']`. The stored relative paths are extended to an absolute path to the file in question which is then added to the archive as relative path again. By doing the path handling this way, it is ensured that paths are always platform independent, as absolute paths are only used when necessary. The `$files_to_store` array, previously `$list_changes['modified']` is dumped as JSON into a temporary file and then stored in the archive as “modified_files.json”. This procedure is also repeated for `$list_changes['remove']` by dumping `$files_to_remove` into another temporary file to build “removed_files.json” and `$database_changes` to “database_changes.json”.

Next, the supplied SQL statements are also dumped into a temporary file to be stored inside of the ZIP archive alongside the previous files as “database_changes.sql” and “database_changes_rollback.sql”:

Figure 37: Connector Backend – Create Patch Package: Database

```
//add SQL as separate file, if supplied
if($sql_statements){
    $temporary_file_sql = sys_get_temp_dir().'\\sql_file';
    file_put_contents($temporary_file_sql, $sql_statements);
    $zip->addFile($temporary_file_sql, 'database_changes.sql');
}
```

Source: Own illustration

After this step, all necessary upgrade information is stored, to enable future delta analysis, however, the entirety of structure arrays needs to be stored too:

Figure 38: Connector Backend – Create Patch Package: Structure

```
//store complete file and database structure information for next delta / integrity checks
//this operation is mandatory
$temporary_file_structure = sys_get_temp_dir().'\\complete_structure';
file_put_contents($temporary_file_structure, json_encode($complete_structure, JSON_PRETTY_PRINT));
$zip->addFile($temporary_file_structure, 'complete_structure.json');

$zip->close();
```

Source: Own illustration

This is done in the same fashion as the previous store procedures. Afterwards the archive is closed and ready to be transferred to the patch library for storage.

Before the actual transfer takes place, an array containing meta data for storage and listing the patch package in the Connector's patch list is created:

Figure 39: Connector Backend – Create Patch Package: Metadata

```
//build meta data from the generated patch package
$meta_data = array(
    'files'      => sizeof($files_to_store),
    'size'       => filesize($zip_file),
    'hash'       => hash('sha256', file_get_contents($zip_file)),
    'optional'   => $optional,
    'name'       => $name,
    'description' => $description
);
```

Source: Own illustration

This is done to quickly access important information about the patch package without extracting it first. Such information is for example the patch packages SHA-256 hash and its name and short description for the list view. This meta data array is then embedded into the actual transfer package which also contains the release note and ZIP archive binary. To ensure a safe transit of this binary data within a data structure, the base64 encoding algorithm is used to convert it to a text first.

Sending of the patch package is done via an HTTP POST request to the Patch Library's "store" endpoint. To do this via PHP, PHP's native implementation of CURL is used.⁶⁸

⁶⁸ CURL is a library to transfer data with URLs, for further information see CURL.

The Patch Library's "store" endpoint then receives the complete package and begins to decode the information contained within to store the patch package in its own database:

Figure 40: Library – Store Endpoint

```
//decode binary data
$release_note_data = base64_decode($release_note_data);
$archive_data      = base64_decode($archive_data);

//extract version and name info from meta data
$meta = json_decode($meta_data, true);

//prepare insert into patches table
$STH = $DBH->prepare('INSERT INTO `patches` (`name`, `version`, `info`, `data`)
                    VALUES (:name, :version, :info, :data)');

$STH->bindParam(':name',    $meta['name']);
$STH->bindParam(':version', $version);
$STH->bindParam(':info',    $meta_data);
$STH->bindParam(':data',    $archive_data, PDO::PARAM_LOB);
If($STH->execute()) {
    $result = 'success';
}else{
    $result = 'failure';
}

//retrieve ID of inserted patch package
$id = $DBH->lastInsertId();
//insert release note
$STH = $DBH->prepare('INSERT INTO `release_notes` (`id`, `language`, `release_note`)
                    VALUES (:id, :language, :release_note)');

$STH->bindParam(':id',          $id);
$STH->bindParam(':language',    $release_note_language);
$STH->bindParam(':release_note', $release_note_data, PDO::PARAM_LOB);
If($STH->execute()) {
    $result = 'success';
}else{
    $result = 'failure';
}
```

Source: Own illustration

Storing the actual patch is a two-step process, as the patch package itself is stored in the `patches` table as described earlier. The second step is storing the release note separately, as this information is language dependent and thus cannot be stored in the same table due to normalization reasons.

The result of the storage procedure is then transmitted back to the Connector's backend as the backends operation is not completely done by sending the package. The Connector's local state needs to be updated to reflect performed changes for further delta analysis and patch installations. This is performed by updating the Connector's system table with the gathered information of the previous steps. The entries "version" and "installed_structure" are updated with the gathered information and the patch packages id, which was returned from the library, is added to "installed_patches", as if it was installed through an installation procedure.

6.3 Installation

Installation of a patch package can be performed by clicking the “install?” button⁶⁹ on an entry in the Connector’s “List of all available patches”. This operation is only possible if the patch was recognized as not already installed by checking its id against a list of all installed patch ids.

The action of clicking said button triggers a call against the Connector’s “apply” endpoint with the patch packages id and hash value, which proceeds as follows:

- **Retrieving the patch package:** The requested patch package is retrieved from the Patch Library’s “retrieve” endpoint by sending the id of the package in question. Transfer of the package is done in the same manner as the previous transfer from the Connector to the Patch Library which was used during the patches creation. To ensure that the file transfer was done correctly and no information was altered, the patch archives SHA-256 hash value is computed again and checked against the value which was transferred together with the id of the package.⁷⁰ The retrieved ZIP archive is then decoded and stored in a temporary location to be accessed by file operations.
- **Extracting the packages contents:** The contents of the patch packages contents are read into arrays for further usage. The files “modified_files.json” and “removed_files.json” are merged into a single \$file_data array in this step, the file “removed_files.json” is also copied into \$file_list_remove to later perform removal of the files it describes. The contents of SQL statement files are written in separate temporary files to access them via MySQL directly, reasoning behind this procedure is described more in depth later when modification of the database is explained.
- **Creating a backup of files:** On the previous extraction stage the array \$files_data was created, it contains the relative paths of all files which are about to change. This array is now used in conjunction with the absolute paths of the application and the backup directory to directly copy files for backup. It is done this way to catch all modifications as well as all files which are removed after applying the patch.

⁶⁹ See appendix 2 for an example of this patch listing.

⁷⁰ See chapter 5.3 regarding usage of hashing.

After these steps have been performed, necessary information for the installation procedure is prepared and the filesystem is ready to be modified, as the backup was just created.

Filesystem changes:

Modification of files is a straightforward task, all files contained in the patch archives “files” subdirectory need to be written to the applications directory. If a file is already present it is simply overwritten:

Figure 41: Connector Backend – File Backup:

```
//entries starting with files/ are file modifications
$modification = 'files\\';
while($zip_read = zip_read($zip_file)) {
    //if the files path begins with files it is an actual file change
    $entry = zip_entry_name($zip_read);
    if(substr($entry, 0, strlen($modification)) == $modification){
        $file_contents = zip_entry_read($zip_read, zip_entry_filesize($zip_read));
        //construct filename for application folder
        $filename = substr($entry, strlen($modification), strlen($entry));
        $filename = $app_path.'\\'.$filename;
        //construct folder name to ensure that folders are correctly set
        $directory = explode('\\\\', $filename);
        array_pop($directory);
        $directory = implode($directory, '\\');
        //create directory if not existing
        if(!is_dir($directory)){
            mkdir($directory);
        }
        //write file to disk, overwrite if necessary
        file_put_contents($filename, $file_contents);
    }
}
```

Source: Own illustration

To do this, the ZIP archive of the patch package is read again. Every entry which is read is checked if its path starts with the “files” subfolder. If it is in this subfolder it is a modification and not a file like the ones which made up the additional information of the patch package like for example “complete_structure.json”.

If this was determined successfully, the absolute path of the file in question in the applications directory is created. Also, its parent directories path is created in the same fashion. Then the directories path is used to check if it is already on the file system, if it is not present, it is created. This measure was taken to accommodate the possibility of adding files within new directories with a patch, as the function “file_put_contents” cannot create directories directly.

After applying the file modifications, the file removal procedure is executed to remove files which are not needed after this patch level. This is done by simply creating the absolute path of every entry in the previously mentioned `$file_list_remove` array and deleting every file found.

Database changes:

To modify the database in a similar fashion as Facebooks OSC library, the following functions were created:

- **create_extra_tables:** This function created two table copies of a given table within a single MySQL transaction. One is named "`<tablename>_mod`" and the other is named "`<tablename>_delta`". This is the preparation for further changes.
- **create_trigger:** This function creates "AFTER INSERT" database trigger statements between a table and its corresponding delta table. This ensures that after calling this function, all inserts to the original are mirrored to the delta table.
- **use_statement:** The "use_statement" function is a shortcut to execute SQL statements directly against the MySQL executable via the command line. This is done to prevent limitations of the PHP database connection, like loading large SQL scripts. It also has the benefit that MySQL treats the contents as a single operation.
- **set_original_data:** Copies data from the original table to its "`_mod`" counterpart. Determination of columns from which to copy and columns which receive values is done by creating the REPLACE statement in conjunction with the previously described SHOW COLUMNS command.
- **set_delta_data:** Copies data from the delta table to its "`_mod`" counterpart in the same fashion as "set_original_data".
- **change_name:** This function swaps the original table with its "`_mod`" counterpart. This is done by first renaming the original to "`<tablename>_mod_temp`", then renaming "`<tablename>_mod`" to original and last renaming "`<tablename>_mod_temp`" to "`<tablename>_mod`".
- **delete_delta:** Removes the "`_delta`" counterpart of a given table.

These functions describe OSC's workflow for modifying a table in a simplified form and are necessary because they have to be applied in loops for every table which is about to be modified. The first step of applying the patch packages SQL statement is then to identify table names which have to be modified and creating the extra tables and triggers required:

Figure 42: Connector Backend – Table Preparation

```
//gather information about needed delta and mod tables
$tables = [];
$tables_failed = -1;
foreach($table_data AS $key => $value){
    $tables[] = $key;
}

//iterate through tables which have to be modified and set triggers
for($i = 0; $i < count($tables); $i++){
    if(create_extra_tables($tables[$i])){
        if(!create_trigger($tables[$i])){
            echo 'set trigger failed for '.$tables[$i];
            $tables_failed = $i;
            break;
        }
    }else{
        $tables_failed = $i;
        echo 'set temp tables failed for '.$tables[$i];
        break;
    }
}
}
```

Source: Own illustration

It is necessary to collect the results of every “create_extra_tables” and “create_trigger” function call, as all operations have to be successful in order to proceed with applying the patches SQL file. Otherwise, the database modification would proceed without backups. If operations failed, the corresponding “_delta” tables are deleted, what remains is the original and its “_mod” counterpart, which can be used to identify why the operation failed.

Afterward, the patch packages SQL file for modifications is applied to the “_mod” tables within the database by calling the “use_statement” function. Then the following functions are then called for every modified table:

- set_original_data
- set_delta_data
- change_name
- delete_delta

This first loads the original tables data into the modified version, then the delta data. Next, it switches the names and cleans up created delta tables.

If this operation was successful for every table, the now called original table is modified with the SQL statements modifications and the “<tablename>_mod” version contains the entirety of the old original table, including its data. After this point, the “_mod” version can be treated as a backup of the original table.

The last operation left is applying changes to the Connector’s internal state storage to record that the patch has been applied and which installed structure is now present in database and filesystem. This is done by loading the patch packages contents of “complete_structure.json” into the Connector’s system table as update for “installed_structure” and updating the “installed_patches” list. This is necessary to be able to provide a starting point for a delta analysis conducted by this Connector and to properly list patch installation states.

7 Testing

This chapter describes the tests which have been carried out to verify that the developed prototype is able to fulfill the goals described and defined in chapter 2.3. As complete testing of the prototype solution is not feasible, and out of the scope of this thesis, first, the boundaries and limitations of the tests are explained. Afterward, the results of the tests are discussed in regard of goal fulfillment.

7.1 Boundaries and Limitations

The basis for the tests is three separate applications which are installed on the same host system to simplify the testing environment. They also use the same web server and MySQL server, installed application setup are as follows:

- **Test application:** The application which is put under control of the Connector for patch management and delta analysis. It consists of its own MySQL database “app” which is accessed by its user “app” and a directory on the filesystem where its application code is stored.⁷¹
- **Connector:** An installation of the prototyped Connector application, installed next to the test application, it also has its own database and database user “connector”. It is located directly next to the “test application” on the filesystem and is able to access the test applications directory. The Connector’s system information is preloaded with the structure of the initial “test application”.⁷²
- **Patch Library:** An installation of the prototyped Patch Library application, it also has its own database, database user and separate filesystem location, communication to this application is to be done through URLs only.⁷³

The initial state of the “test application” is a basic filesystem and database layout as follows:⁷⁴

⁷¹ The test application is contained within the “test_application” directories in the digital appendix.

⁷² The Connector is contained within the “connector” directory in the digital appendix.

⁷³ The Patch Library is contained within the “library” directory in the digital appendix.

⁷⁴ See digital appendix “appendix/tests/basis/” for exact layout of the example application.

- **Filesystem:** Three folders which contain a variety of different text files. To check the Connector's ability to scan subfolders, one folder contains another directory which in turn contains more files.
- **Database:** The database of the test application consists of three tables with no foreign key constraints, column names are not database unique to test if the column changes are recognized properly.

To test the defined goals, the following tests will be conducted:

Table 10: Tests

Id	Test case	Description
1	File add detection	Test to check if the prototype correctly identifies files which have been added to the application.
2	File removal detection	Test to check if the prototype correctly identifies removed files and stores this information correctly.
3	File update detection	Test to check if changes to already existing files are recognized and handled properly.
4	Patch package creation	Test to check if creation and storage of a complete patch package were successful and if it contains all necessary data.
5	Correct patch listing	Test to check if retrieval of the patch list from the Patch Library and matching against the Connector's internal system information is correct.
6	Correct patch delivery	Test to check if the prototype recognizes if the SHA hash of a patch package has been modified.
7	Filesystem backup	Test to check if the filesystem backup was done correctly regarding removed files and changes.
8	Filesystem updates	Test to check if updates to the filesystem were applied properly.
9	Database backup	Test to check if, after an applied database change, the "_mod" table is containing a backup of the old version.
10	Database updates	Check if a database change is carried out correctly

Source: Own illustration

Every test will be conducted in a self-contained fashion, that means that after every test all three applications are returned to their respective initial state, the Patch Library may be preloaded with a patch package if the test requires one. Some tests require tampering with the database directly to measure results, without installing a second Connector or disrupting file transmission on purpose.

These special cases are: test 5: “correct patch listing”, as the Connector updates its own stored patch history after the creation of a patch and thus is unable to create patches which are in an “uninstalled” state for itself. The other special case is test 6: “correct patch delivery”, corrupting the ZIP archives data on purpose through network related tools seems unfeasible. However, the patch packages stored SHA-256 value can be modified to simulate that behavior.

After each performed step, relevant information is gathered from every application. For all patch creation related tests that means that the resulting ZIP archive is stored, as well as the produced meta data information which is retrieved from the patch library. For all tests which modify the filesystem, the changed application source is stored before creating the patch package. For database changes however, the tables are exported through the database administration tool “phpMyAdmin” as SQL statement files. Saving the database files directly would mean collecting binary data which cannot easily be viewed or checked for correctness.

7.2 Results

The results were mostly positive, only one test case was performed unsuccessfully:

Table 11: Test Results

Id	Test case	Result
1	File add detection	Successful, the two added files were correctly stored within “modified_files.json”.
2	File removal detection	Successful, three files were removed and properly written to “removed_files.json”
3	File update detection	Unsuccessful, the changed file was recognized as a modification as well as a removed file.
4	Patch package creation	Successful, files were correctly recognized as added or deleted.
5	Correct patch listing	Successful, Patch listing was appended with locally stored information.
6	Correct patch delivery	Successful, the patch list included an SHA hash which was different from the patch packages own (the first character was changed), installation canceled.
7	Filesystem backup	Successful, all three files to removed were first copied to the backup location.
8	Filesystem updates	Successful, updates were done correctly.
9	Database backup	Successful, “users_mod” contained all original data.
10	Database updates	Successful, “users” was altered correctly.

Source: Own illustration based on performed tests⁷⁵

Most of the detections ran flawlessly, detected and stored all information necessary. However, test 3 proved to be unsuccessful, the changed file was identified as removed and as a modification. This issue has been mitigated for file installation by removing files first and then copying over the changes. As such it is only a minor bug.

⁷⁵ See digital appendix “/tests/” for further information.

In regard to the defined goals in chapter 2.3 the following goals were achieved:

- **Patch dependencies:** This goal described an automatic handling of patch dependencies, this is implemented in a basic form as version numbering. Incrementing version numbers is done automatically and the patch listing checks if patches were already installed by checking against a list of installed patches. If a patch is deemed already installed, the Connector provides no means to install it again. Only patches which have not yet been installed are available for installation.
- **Patch Library:** The goal was to provide a centralized platform which stores all necessary, patch package related, data. This was achieved partially with the Patch Library application. Patches and their release notes are stored correctly. However localization of release notes is not entirely implemented, the Patch Library's database structure allows it, but the Connector does not currently allow to choose the language.
- **Patch construction:** This goal was achieved; the Connector collects all necessary data to build a complete patch package and is able to perform construction of a package. Delta analysis is done completely automated, as is a collection of required files and files to remove. A minor bug currently causes collection of changes as a file removal and change. However, the SQL statements for database updates need to be supplied manually.
- **Patch installation:** This goal was to provide means to implement the contents of a patch package in an entirely automated manner. As the Connector is able to connect to the Patch Library, retrieve a selected patch package and afterward, update the filesystem and database with a patch package without the need for human intervention this goal is also achieved.

8 Estimated Impact

This chapter focuses on the impact of implementing an automated patch deployment system in the company on which the case study is based. As such the following estimates are based on the experience of this company in the deployment of its existing solutions. Therefore, the amount of work required to deliver a patch is well known. The impact of using an automated patch deployment and control system was estimated very carefully and conservatively. This, however, does not guarantee or claim that the calculations are representative for a wider audience. To further diversify discussion about the impact of using an automated patch deployment and control system, two aspects will be treated separately; quantitative impact and qualitative impact.

8.1 Quantitative Impact

To measure the quantitative impact of using the automation of the case study's workflow, the effort of using the workflow manually needs to be taken into consideration. The following distinctions were made regarding types of patches:

- Major updates: significant changes or addition to features provided by the application. This often includes drastic changes to files and database.
- Minor updates: small changes in functionality, additions to existing features.
- Bugfixes: minor changes to correct known malfunction of the application, these do not alter features.

General effort is also required on a per-patch basis, regardless of the type of patch. This effort is called "Creation" and describes the necessary work to be done when creating a patch package. This essential work is mostly made up of delta analysis and description of the installation procedure. The following table of needed effort describes the work to be done by the software supplier and the customer regarding different types of patches:

Table 12: Patch Effort:

Task	Effort Supplier (in man-days)		Effort Customer (in man-days)	
	Minimum	Maximum	Minimum	Maximum
Major Update	0.5	1	1	2

Minor Update	0	1	0.5	0.5
Bugfix	0	0.25	0.5	0.5
Creation	1	2	-	-

Source: Own illustration based on experience⁷⁶

Effort on the supplier side consists of test deployments, which depend on the type of patch. Larger patches usually require a more complicated installation during which one employee performs installation per release note and another check if all file and database updates are correctly carried out. Regardless of these test deployments, however, manual creation of a patch package is still required for every patch and consists of manually performing delta analysis. The longer the needed installation procedure, the longer it also takes to create a release note which describes the necessary steps to install.

Effort on the customer's side is higher for different types of patches due to fulfilling documentation guidelines. This might have an enormous impact, depending on jurisdiction and regulation e.g. in the financial industry. The larger the installation procedure, the larger the needed documentation, as every manual step has to be documented. This procedure has to be repeated for the productive system as well, after all tests on the test system have been completed.

The estimated amount of effort after implementing an automated patch deployment and control system is as follows:

Table 13: Patch Effort Adjusted:

Task	Effort Supplier (in man-days)		Effort Customer (in man-days)	
	Minimum	Maximum	Minimum	Maximum
Major Update	0	0.5	1	2
Minor Update	0	0	0.25	0.25
Bugfix	0	0	0.5	0.5
Creation	0.5	1	-	-

Source: Own illustration based on estimates⁷⁷

⁷⁶ The data of this table is based on the experience of the managing director of fidis GmbH and is by no means representative for general software maintenance effort as it is based on a single company's empirical data and processes.

⁷⁷ The data of this table is based on the estimates of the managing director of fidis GmbH.

The supplier's reduction in effort is mainly due to the elimination of manual delta analysis as that is the most error-prone part of the patch creation step. Required manual effort is only left in checking the completed patch package and creation of SQL statements which is also assisted by showing all changes which need to be taken into consideration.

The scaling of manual installation is no longer given, as the installation part is also completely automated. The needed effort in finding errors in a failed installation is also tremendously reduced as the system keeps track of version numbers, which proved to be a problem according to the case study. This also usually required another manual delta analysis to identify what caused the installation failure.⁷⁸ These reductions in the effort are also reflected on the customer's side as only the necessary effort for documentation are left. All manual tasks which needed to be done are automated, only their results need to be checked and documented according to the customer's requirements.

8.2 Qualitative Impact

The qualitative impact is not directly measurable by just changing the used software maintenance workflow to an automated one as provided by the prototype. To accurately measure and analyze a possible improvement in software quality, measurement KPIs⁷⁹ would have to be defined in accordance with ISO 25010, which describes software quality and how to increase it.

However, it is still possible to draw conclusions to a company's software quality as the following benefits can be derived from the error mitigation of using the prototype, even without accurately defined KPIs:

- Patch creation: An automated delta analysis covers the entirety of filesystem and database changes, as such, there can be no changes which are not identified upon creation of a patch package. The automated patch packaging also ensures that all necessary modifications are included in a package. The automation leaves no room for missed files or similar.
- Patch delivery: This part had several spots where errors could occur: previously patch packages were sent to administrators before the patch was installed to an application. This lead to a necessary overhead on the administrator's side to store

⁷⁸ See chapter 2.2.

⁷⁹ Key Performance Indices (KPI) are used to measure e.g. quality by a defined set of rules.

patches and apply them in the correct order. As sending patch packages to administrators for future usage is now a step which is removed from the workflow and sending only occurs on-demand, the probability of missing a critical patch is removed. The Connector also keeps track of already installed patches and compares them with a list of patch packages from the patch library. This entirely avoids the error-prone parts of the manual workflow of patch delivery.

- Patch installation: The installation procedure does not need a manual detection of tables or files to backup for a rollback anymore as both steps are now automated. In addition, the installation itself is also automated. Both automatizations reduce the possibility of errors.

These benefits ensure that the influence of errors for measuring the quality of a software product supported with an automated maintenance solution like the prototype is greatly reduced. And thus, quality regarding error-free deployment and is increased, which naturally also increases user satisfaction due to smaller downtimes in which the application cannot be accessed. This also leads to the conclusion that errors which are encountered in the application may be due to an actual bug in the software and not due to an unstable application, caused by a faulted patch installation. This conclusion was an estimated guess at best, due to the error-prone nature of the manual process. After implementing automation, it is most likely the case that an error in the application is an actual bug and not a side-effect of a faulted installation.

9 Conclusion

The focus of this thesis was to optimize the vital process of patch deployment for software products to a point where fragile, repetitive parts are solely performed by a prototyped, specialized software. The only choices left to a human operator should be where and when a patch will be deployed, not which technical measures need to be taken to do so. Boundaries of the work implicated that the software implementation was restricted to a prototype which uses a standard LAMP application and a specific workflow.

9.1 Summary

The process for creating, distributing and installing patches for the case study's application was outlined and could be viewed as three separate workflows:

- Construction of a patch package
- Delivery of a patch package
- Installation of a patch package

The case study's workflows formed the basis for a streamlined process of patch handling but also showed the weaknesses of each separate workflow and probability of error. Through this examination of the workflows and their respective error spots goals could be derived which would mitigate or outright eliminate errors.⁸⁰ Further examination of the most repetitive and error burdened parts: detecting changes in the filesystem and database lead to an evaluation of existing solutions for these types of tasks.⁸¹

Following the evaluation, a generalized solution encompassing the previously defined goals and the study's workflow was designed.⁸² This software solution was designed to consist of two separate applications: a "Patch Library" which stores all patches and handles delivery and storage and a patch "Connector", which handles construction and installation of patch packages. As both applications could be built with the same technology that made up the target application which needs patch handling, there was an op-

⁸⁰ See Chapter 2.2.

⁸¹ See Chapter 4.

⁸² See Chapter 5.1 - 5.2.

portunity to significantly reduce the footprint of the Connector application. This opportunity was to reuse the technology stack of the case study's application. This led to a concept that would use the target applications MySQL server as well as the PHP runtime environment. Following this initial platform choice, a generalized abstraction of the case study's workflows and necessary information storage were designed. The evaluated software solutions were discussed regarding implementation afterward. A closer examination of the existing solutions revealed that implementation of source control solutions like SVN or Git to synchronize files leads to the dependency of installing a separate client program for those solutions and thus a heavily increased footprint of the Connector.⁸³ This led to the creation of a prototype which would only recycle parts of discussed solutions and otherwise implement the abstracted algorithms which automated the workflows of the case study.⁸⁴

Afterward, necessary tests for the prototype were defined. These tests were also based on the goals which were defined earlier and aimed to verify the automation of the case study's workflow.

The tests showed that the prototype was fit for its purpose, only a minor bug was found which could be mitigated.

After testing the prototype, the impact of using such a system was discussed from a quantitative and qualitative view. This was done to reflect which economic value an automated workflow and usage of such a system could bring for a company which uses it. As such the discussion was made with the data from the company which provided the case study's workflow.

In conclusion, it can be said that conception and prototyping of a mostly automated patch deployment and control system were successful in regard to the defined goals and the added economic value for a company is significant.

⁸³ See Chapter 5.3.

⁸⁴ See Chapter 6.

9.2 Outlook

While conception and prototyping were successful, there is still plenty of room for improvements. For example the prototypes GUI was not designed for actual usage but mainly with functionality in mind. As such it could be improved by showing a graphical view of delta changes to provide a more accessible means to verify recognized changes.

Another improvement would be to entirely automate the process of database alteration by generating the needed SQL statements instead of just showing the changes that have been made and prompting the patch creating user for SQL statements to modify the database and to revert them.

Also, the prototype provided just the basic functionality to automate the workflow. The tests showed that there is plenty of room for improvement in handling side-tasks which affect the workflow, like handling backups. Currently the prototype only produces backups, but otherwise, it does not show which files and database tables are currently held in a backup state, handling these is left to the Connector's user. It also gives no actions to interact with these files.

The concept of the Patch Library itself can also be greatly expanded, as it is currently nothing more than a remote database access point. The prototype was designed to handle a single application's patches to test the concept. For real-world usage, however, the library should be able to support storing patches for different applications. This addition then would also require a GUI for the library itself. Also, authentication and verification between both applications are entirely absent in the prototype. This also needs to be added, as the blind trust between two remote servers is a security risk.

While the prototype automatically increases the application's version number with each created patch package, there is room for improvement to let the solution handle version numbering in a semantic way entirely. The chapter 8 "Estimated Impact" differentiates between different types of patches. This can be used to automate version numbering, for example for a version number definition like <major version>.<minor version>.<bug-fix>, e.g. 1.12.8. This could be realized by adding the functionality of selecting which type of patch was created during the Connector's patch creation process; the version number could then be incremented accordingly.

Appendix

List of Appendices

Appendix 1: JSON Structure for an entire database table	79
Appendix 2: Connector Patch Listing:	80
Appendix 3: enclosed CD containing electronic versions of sources	81

Appendix 1: JSON Structure for an entire database table

```

1 {
2   "modified": {
3     "users": [
4       {
5         "name": "id",
6         "type": "int(11)",
7         "collation": null,
8         "null": "NO",
9         "key": "PRI",
10        "default": null,
11        "extra": "auto_increment"
12      },
13      {
14        "name": "username",
15        "type": "varchar(40)",
16        "collation": "latin1_swedish_ci",
17        "null": "NO",
18        "key": "",
19        "default": null,
20        "extra": ""
21      },
22      {
23        "name": "password",
24        "type": "varchar(80)",
25        "collation": "latin1_swedish_ci",
26        "null": "NO",
27        "key": "",
28        "default": null,
29        "extra": ""
30      },
31      {
32        "name": "name",
33        "type": "varchar(60)",
34        "collation": "latin1_swedish_ci",
35        "null": "NO",
36        "key": "",
37        "default": null,
38        "extra": ""
39      },
40      {
41        "name": "settings",
42        "type": "text",
43        "collation": "latin1_swedish_ci",
44        "null": "NO",
45        "key": "",
46        "default": null,
47        "extra": ""
48      }
49    ]
50  },
51  "removed": []
52 }

```

Source: Own illustration

Appendix 2: Connector Patch Listing:

PATCHES - LIST		
List of all available patches		
Initial Creation Package	installed	This is the initial creation package for the test application.
Test 1	installed	Test 1 is done to check correct identification of added files
Test 2	installed	Test 2 is done to check correct identification of removed files
Test 3	installed	Test 3 is done to check if a file change is correctly handled
Test 4	installed	Test 4 is used to test creation of a complete patch package which contains all types of changes.
Testpatch Database Mod	install?	This is a patch package to test database modification and backup abilities.
		<i>i</i> Release note
		<i>i</i> Release note
		<i>i</i> Release note
		<i>i</i> Release note
		<i>i</i> Release note

Source: Own illustration

Appendix 3: enclosed CD containing electronic versions of sources

Full list of contents:

- Electronic versions of online sources
- Electronic version of this Thesis
- Source code of a test application
- Source code of the prototypes Connector application
- Source code of the prototypes Connector application
- SQL statements to create the database for the test application
- SQL statements to create the database for the Connector
- SQL statements to create the database for the Patch Library
- Snapshots of all data gathered during the tests
- Initial versions of all applications already prepared for tests

List of Cited Literature

Books

Balasubramaniam, S/ Pierce, Benjamin C. (1998): Balasubramaniam, S; Pierce, Benjamin C: What is a File Synchronizer?. SCSI Technical Report #507, Indiana University, April 22, 1998.

Chen, Lianping/ Power, Paddy (2015): Chen, Lianping; Power, Paddy: Continuous Delivery Huge Benefits, but Challenges Too, IEEE Software, March/April 2015.

Farley, David/ Humble, Jez (2010): Farley, David; Humble, Jez: Continuous delivery: reliable software releases through build, test, and deployment automation, Pearson Education, Boston, August 2010.

Hines, Peter/ Rich, Nick (1997): Hines, Peter; Rich, Nick: The seven value stream mapping tools, In International Journal of Operations & Production Management, Vol. 17 Issue 1, 1997.

Huffman, David A. (1952): Huffman, David A.: A Method for the Construction of Minimum-Redundancy Codes, In Proceedings of the IRE, Vol. 40, Issue 9, September 1952.

ISO14764 (1999): International Standard ISO/IEC1474: Information technology – Software maintenance, first edition, 15 November 1999.

Lempel, Abraham/ Ziv, Jacob (1977): Lempel, Abraham; Ziv, Jacob: A Universal Algorithm for Sequential Data Compression, In IEEE Transactions of Information Theory, Vol. IT-23, Issue 3, May 1977.

Loeliger, Jon (2009): Loeliger, Jon: Version Control with Git, O'Reilly Media, first edition, May 2009.

Office of Information Services (2008): Centers for Medicare & Medicaid Services (CMS): Selecting a Development Approach, 17 February 2005.

Pierce, Benjamin C./ Vouillon, Jérôme (2004): Pierce, Benjamin C.; Vouillon, Jérôme: What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer, Technical Report MS-CIS-03-36, Department of Computer and Information Science, University of Pennsylvania, 24 February 2004.

Skelton, Matthew/ O'Dell, Chris, (2016): Skelton, Matthew; O'Dell, Chris: Continuous Delivery with Windows and .NET, O'Reilly Media, first edition, 25 February 2016.

Internet Documents

(All internet documents were last checked on 8 May 2017)

Ambler, Scott W. (2012): Ambler, Scott W.: The Agile System Development Life Cycle (SDLC), Online, URL: <http://www.ambysoft.com/essays/agileLifecycle.html>.

Callaghan, Mark (2010): Callaghan, Mark: Online Schema Change for MySQL, Online, URL: https://www.facebook.com/note.php?note_id=430801045932.

Collins-Sussman, Ben/ Fitzpatrick, Brian W./ Pilato, C. Michael (2011): Collins-Sussman, Ben; Fitzpatrick, Brian W.; Pilato, C. Michael: Version Control with Subversion, Online, URL: <http://svnbook.red-bean.com/en/1.7/>.

Crockford, D. (2006): Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON), Online, URL: <https://tools.ietf.org/html/rfc4627>.

CURL: curl: command line tool and library, Online, URL: <https://curl.haxx.se/>.

DB-Engines (2017): solid IT gmbh, DB-Engines Ranking – die Rangliste der populärsten Datenbankmanagementsysteme, Online, URL: <https://db-engines.com/de/ranking>.

Deutsch, P. (1996a): Deutsch, P.: DEFLATE Compressed Data Format Specification version 1.3, Aladdin Enterprises, Online, URL: <https://tools.ietf.org/html/rfc1951>.

Deutsch, P. (1996b): Deutsch, P.: GZIP file format specification version 4.3, Aladdin Enterprises, Online, URL: <https://tools.ietf.org/html/rfc1952>.

Dougherty (2001): Dougherty, Dale: LAMP: The Open Source Web Platform, Online, URL: <http://www.onlamp.com/pub/a/onlamp/2001/01/25/lamp.html>.

Facebook OSC: Online Schema Change aka OSC, Online, URL: <http://baazaar.launchpad.net/~mysqlatfacebook/mysqlatfacebook/tools/annotate/head:/osc/OnlineSchemaChange.php>.

Git – Basics: Git Basics – Recording Changes to the Repository, Online, URL: <https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>.

Git-Tower (2017): fournova Software GmbH: Learn Version Control with Git, Online, URL: <https://www.git-tower.com/learn/git/ebook/en/command-line/remote-repositories/introduction>.

GitHub (2017): GitHub – The world's leading development platform, Online, URL: <https://github.com/>.

Hansen, T. (2011): Hansen, T.: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF), Huawei, Online, URL: <https://tools.ietf.org/html/rfc6234>.

Huffman Tree Generator: Ligus, Slawek; Huffman Tree Generator, Online, URL: <http://huffman.ooz.ie/>.

Humble, Jez (2016): Humble, Jez: Patterns – Continuous Delivery, Online, URL: <https://continuousdelivery.com/implementing/patterns/>.

Intel (2013): Intel: Intel SHA Extensions, Online, URL: <https://software.intel.com/en-us/articles/intel-sha-extensions>.

Jones, P. (2001): Jones, P.: US Secure Hash Algorithm 1 (SHA1), Motorola, Online, URL: <https://tools.ietf.org/html/rfc3174>.

MySQL - SHOW COLUMNS: MySQL 5.7 Reference Manual – SHOW COLUMNS Syntax, Online, URL: <https://dev.mysql.com/doc/refman/5.7/en/show-columns.html>.

MySQL - SHOW TABLES: MySQL 5.7 Reference Manual – SHOW TABLES Syntax, Online, URL: <https://dev.mysql.com/doc/refman/5.7/en/show-tables.html>.

Netcraft (2014): Netcraft Ltd.: June 2014 Web Server Survey, Online, URL: <https://news.netcraft.com/archives/2014/06/06/june-2014-web-server-survey.html>.

PHP – JSON: PHP: Requirements – Manual, Online, URL: <http://php.net/manual/en/json.requirements.php>.

PHP – XML: PHP: Requirements – Manual, Online, URL: <http://php.net/manual/en/xml.requirements.php>.

Ruby on Rails – AR: Active Record Basics – Ruby on Rails Guides, Online, URL: http://guides.rubyonrails.org/active_record_basics.html.

Ruby on Rails - AR Migrations: Active Record Migrations – Ruby on Rails Guides, Online, URL: http://edgeguides.rubyonrails.org/active_record_migrations.html.

Shattered.it (2017): SHattered, Online, URL: <http://shattered.it/>.

SoundCloud LHM (2016): soundcloud/lhm: Online MySQL schema migrations, Online, URL: <https://github.com/soundcloud/lhm>.

Subversion (2016): Apache Subversion, Online, URL: <https://subversion.apache.org/>.

W3C – XML: Extensible Markup Language, Online, URL: <https://www.w3.org/XML/>.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Masterthesis selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form ganz oder teilweise noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 08.05.2017

Ort, Datum

Unterschrift