BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

IMPLEMENTATION AND EXTENSION OF THE TECHNICAL DOCUMENTATION TESTING FRAMEWORK

IMPLEMENTACE A ROZŠÍŘENÍ FRAMEWORKU PRO TESTOVÁNÍ TECHNICKÉ DOKUMENTACE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Peter Macko**
AUTOR PRÁCE

SUPERVISOR — **Ing. Petr Ilgner**
VEDOUCÍ PRÁCE

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Peter Macko ***ID:*** 164326
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Implementace a rozšíření frameworku pro testování technické dokumentace

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce je vytvoření uceleného frameworku určeného k testování technické dokumentace psané ve značkovacím jazyce AsciiDoc. V rámci semestrální práce student popíše specifika tvorby technické dokumentace a formáty používané pro tvorbu strukturované technické dokumentace, popíše význam sémantického značkování a průběžné integrace (CI) v kontextu tvorby technické dokumentace. Dále se seznámí se s možnostmi nabízenými systémem Emender, který je určený pro deklaraci a spouštění testů nad dokumenty. Student seznámí s metodami pro identifikaci a označení “false positives” ve výsledcích testů, tzv. “waiver”, a pro manuální “odmávaní” chybných výsledků testů. V praktické části student navrhne strukturu databáze vhodné pro uložení informací získaných od uživatele a implementuje framework Emender na dokumentační set psaný ve formátu AsciiDoc. V diplomové práci student implementuje do systému Emender webovou službu s REST API komunikující s databází. Pro implementaci použije jazyk Python a vhodný webový framework (např. Flask). Všechny testy budou podporovat formát AsciiDoc. Dále popíše možnosti prezentace výsledků testů a navrhne způsob navázání testovacího frameworku na systémy pro zajištění průběžné integrace (např. systém Jenkins). Výsledky testů na zvolených technických dokumentech vhodným způsobem prezentuje a okomentuje. Na závěr student navrhne možnosti dalšího vývoje projektu.

DOPORUČENÁ LITERATURA:

[1] HILLAR, Gastgon. C. Building RESTful Python Web Services. Paperback. United Kingdom: Packt Publishing - ebooks Account, 2016. ISBN 978-1786462251.

[2] GRINBERG, Miguel. Flask Web Development, 2nd Edition. 2. United Kingdom: O'Reilly Media, 2018. ISBN 9781491991725.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. Petr Ilgner
Konzultant: Pavel Tišnovský, Red Hat Czech s.r.o.

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRACT

The thesis discusses automated testing of technical documentation written in AsciiDoc markup language using open-source documentation testing framework Emender implemented in CI/CD. The framework was extended with a RESTful web application emenderwebservice, providing a graphical user interface with test results and a mechanism to waive false positive test results. Web application was implemented with Flask WSGI web application framework along with a database enabling aggregation and unique test identification. The application simplifies access to test results generated by Emender in CI/CD and provides a concise graphical user interface for technical writers.

KEYWORDS

Automated testing of technical documentation, documentation testing framework Emender, AsciiDoc, Jenkins, CI/CD, False positive results in technical documentation testing, Flask

ABSTRAKT

Práca sa zaoberá automatizáciou testovania technickej dokumentácie napísanej v značkovacom jazyku AsciiDoc pomocou open-source frameworku testovania technickej dokumentácie Emender implementovaného na CI/CD platforme. Framework bol rozšírený o webovú aplikáciu emenderwebservice s REST API, ktorá poskytuje užívateľské grafické rozhranie s výsledkami testov a mechanizmom na odrieknutie falošne pozitívnych výsledkov testov. Webová aplikácia bola vytvorená pomocou WSGI frameworku na tvorbu webových aplikácií Flask s databázou ktorá umožňuje agregáciu výsledkov testov a ich unikátnu identifikáciu. Aplikácia uľahčuje prístup ku výsledkom testov vygenerovaných frameworkom Emender v CI/CD systémoch a poskytuje technical writer-om ucelené užívateľské prostredie.

KLÍČOVÁ SLOVA

Automatizácia testovania technickej dokumentácie, framework na testovanie technickej dokumentácie Emender, AsciiDoc, Jenkins, CI/CD, falošne pozitívne výsledky pri testovaní technickej dokumentácie, Flask

MACKO, Peter. *Implementation and Extension of Technical Documentation Verification Framework*. Brno, Rok, 67 p. Master's Thesis. Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Advised by Ing. Petr Ilgner

ROZŠÍŘENÝ ABSTRAKT

Práca pojednáva o tvorbe technickej dokumentácie, na ktorú sa posledné roky v rámci vývoja softvéru kladie stále väčší dôraz. Technická dokumentácia slúži ako jedno z premostení medzi vývojárom softvéru a koncovým používateľom. Pri čoraz komplexnejších softvérových implementáciách je preto nevyhnutnou súčasťou dodávaného softvéru.

Fázy procesu tvorby technickej dokumentácie sú podobné ako pri vývoji softvéru – dizajn, implementácia a verifikácia. Jedna z úloh autora technickej dokumentácie (Technical Writer) je na seba prevziať rolu koncového zákazníka. Tento pohľad umožňuje vývojárom mimo iného identifikovať nedostatky v technickej stránke dokumentácie, ktoré mohli byť počas vývoja prehliadnuté.

Vo fáze verifikácie technickej dokumentácie sa vyskytujú úlohy, ktoré sú efektívne automatizovateľné – napríklad testovanie funkčnosti hypertextových odkazov, výskytu zakázaných slov alebo ľubovoľných atribútov definovaných značkovacími jazykmi. Pre tento účel bol vyvinutý framework Emender, ktorý poskytuje platformu na písanie testov technickej dokumentácie v jazyku Lua. Na tomto frameworku boli implementované testy použité v praxi, ako napríklad:

- testovanie funkčnosti odkazov,
- testovanie balíčkov, pri ktorom sa overí, či daný balíček patrí danej distribúcii softvéru (Linux),
- testovanie, či dokument neobsahuje špeciálne znaky po zlučovaní vetiev (merge) pomocou systému riadenia revízií Git,
- testovanie štylistiky,
- testovanie korektného nastavenia atribútov dokumentu, a iné.

Emender poskytuje výsledky testov v niekoľkých formátoch – JSON, JUnit XML, HTML a čistý text. Testy môžu byť spúšťané lokálne na vyžiadanie, ale po ich implementácii do systému Jenkins CI/CD aj automaticky pri zmene dokumentácie. Pre tento spôsob použitia má však aktuálna implementácia frameworku Emender isté nedostatky. Medzi ne patrí napríklad nutnosť poznať systém Jenkins CI/CD a jeho štruktúru. Ten navyše zastrešuje veľa úloh, čím sa stáva pomalý a neprehľadný. Výsledky testov sú dostupné len pod číslom zostavenia (buildu) v Jenkinse, čo znemožňuje ich jednouchú identifikáciu.

Niektoré testy môžu už od svojho návrhu hlásiť chybové výsledky. Dôvodom môže byť nesprávny návrh testu, ale tiež sa môže jednať o jeho žiadúcu vlastnosť – chybový výsledok môže byť použitý na získanie pozornosti používateľa na danú časť dokumentácie, ako napríklad použitie nevhodného (i keď funkčného) formátu hypertextového odkazu. Framework Emender neobsahuje mechanizmus, ktorým by mal používateľ možnosť takéto výsledky filtrovať.

Hlavným cieľom tejto diplomovej práce bolo navrhnutie webovej aplikácie emenderwebservice poskytujúcej REST aplikačné rozhranie (API). Táto aplikácia poskytuje výsledky testov v prehľadnom grafickom rozhraní, pričom obsahuje mechanizmus na spracovanie falošne pozitívnych (false positive) výsledkov testov agregovaných v databáze SQLite. Aplikácia bola navrhnutá tak, aby bola spätne kompatibilná s existujúcou implementáciou frameworku Emender v Jenkins CI/CD.

Stručné zhrnutie funkčnosti navrhnutej aplikácie v existujúcej CI/CD implementácii:

1. Používateľ spraví zmenu v dokumentácii uloženej v Git repozitári.
2. Táto zmena spustí testovanie novej revízie dokumentácie v systéme Jenkins.
3. Výsledky testov vo formáte JSON odošle Jenkins na REST API aplikácie emenderwebservice, ktorá výsledky identifikuje podľa parametrov v URI a uloží ich do databázy.
4. Na rovnakej URI aplikácia poskytne grafické rozhranie, čím sa zabezpečí jednoduchá identifikácia testov. Toto grafické rozhranie poskytuje mechanizmus na spracovanie chybných výsledkov.
5. Webová aplikácia vygeneruje a poskytne systému Jenkins súbor JUnit XML, ktorý Jenkins spracuje za účelom vyhodnotenia trendov výsledkov testov v grafickej podobe.

Pri implementácií navrhnutej webovej aplikácie emenderwebservice boli využité nasledujúce technológie:

- WSGI (Web Server Gateway Interface) framework Flask pre vytvorenie REST aplikačného rozhrania,
- šablónový procesor Jinja zahrnutý vo frameworku Flask pre vytvorenie grafického rozhrania,
- relačný databázový systém SQLite3 pre agregáciu testov.

Aplikácia je postavená na návrhovom vzore MVC (Model-View-Contoller), pričom:

- „Model“ predstavuje databáza, ktorej štruktúra bola navrhnutá tak, aby vhodne reprezentovala unikátne výsledky testov,
- „View“ predstavuje Jinja šablóna, prostredníctvom ktorej sú dynamicky generované výsledky testov s mechanizmom na spracovanie falošne pozitívnych výsledkov,
- „Controller“ predstavuje jednak poskytované REST aplikačné rozhranie, ale tiež časti aplikácie zabezpečujúce jej riadenie a spracovanie výsledkov.

Spracovanie falošne pozitívnych výsledkov je zabezpečené tak, že používateľ má možnosť takéto výsledky identifikovať prostredníctvom grafického rozhrania. Označením výsledku ako falošne pozitívneho sa spustí JavaScript kód, ktorý vykoná AJAX volanie na REST API aplikácie emenderwebservice. Toto volanie obsahuje informáciu o označení výsledku spolu s jeho identifikátorom. Aplikácia po prijatí požiadavky cez REST API následne túto zmenu uloží do databázy.

Záver

Implementácia webovej aplikácie emenderwebservice do frameworku Emender poskytuje jednoduché používateľské prostredie a jednotnú agregáciu výsledkov testov prístupnú cez REST API. Unikátna identifikácia testov je zabezpečená použitím čitateľných unikátnych URI parametrov, ktoré poskytujú informácie o danej dokumentácii a použitom teste. Emender je open-source projekt, ktorý našiel svoje uplatnenie v praxi, avšak jeho nedostatky v podobe nedostatočnej dokumentácie a neoptimálnej prezentácie výsledkov testov môžu brániť jeho širšiemu nasadeniu.

Táto práca sa usiluje niektoré tieto nedostatky odstrániť, mimo iného poskytnutím uceleného prehľadu o frameworku a implementácií užívateľsky príjemnej prezentácií výsledkov testov.

Prohlášení

Prohlašuji, že svou diplomovou práci na téma „*Implementace a rozšíření frameworku pro testování technické dokumentace*“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následku porušení ustanovení § 11 a následujících autorského zákona c. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonu (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne 1.6.2020 .....

podpis autora

DECLARATION

I declare that I have written the Master's Thesis titled "Implementation and Extension of Technical Documentation Verification Framework" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

I'd like to thank Ing. Petr Ilgner and Ing. Pavel Tišnovský, PhD. for the supervision of my thesis. I'd also like to thank my family and friends for the unconditional support they provided during the writing of the thesis.

Brno ....

author's signature

Contents

List of Figures

Listings

Introduction

With growing complexity of the software, the importance of technical documentation is becoming more significant. What was once a responsibility of a software developer is now steadily being transferred onto dedicated technical writer teams.

The phases of writing technical documentation can be in many aspects compared to software development - design, implementation, verification. The stages for creating enterprise documentation are similar and equally important.

The documentation testing framework Emender[1] was developed in cooperation with technical writer teams as an effort to automate documentation testing tasks that can be completed more quickly and precisely by software, reducing writing time and mistakes from manual verification.

The thesis discusses aspects of creating technical documentation written in AsciiDoc markup language and testing automation by CI/CD applications. The goal is to extend the open-source documentation verification framework Emender with a RESTful web application enabling presentation of test results through a graphical user interface with a mechanism for tagging false-positives in test results, colloquially "waiver".

The text is structured into several sections:

- Introduction to methods and technologies used for writing technical documentation (Semantic markup languages, rendering and publication tools, CI (Continuous Integration) systems),
- Introduction to documentation testing framework Emender and its implementation on technical documentation,
- Research of possibilities to extend the Emender framework with a RESTful web application providing graphical user interface providing test results with a mechanism to waive false positive results,
- Implementation of the web application using a WSGI Python framework Flask, complemented by SQLite3 database enabling aggregation of test results,
- Suggestions for a further development of the Emender project.

[1] https://github.com/emender/emender

1 Technical Documentation

The term *technical documentation* includes multiple types of documents that describe product functionality and its properties. Technical documentation in context of software development serves as a bridge between the software user and a developer enabling to present the software in a comprehensible manner.

The role of technical writer is to understand the needs of readers and to communicate these to the developer teams, by working with provided drafts, transforming these into customer content, continuously testing and providing feedback and requests for technical validation to another teams.

As stated in the introduction, the process of writing technical documentation is similar to software development, and it is customary to have technical writers as a part of developer team, not as an independent entity. This enables writers to be a part of the software development, and to create the documentation in iterations along with it. The integration of writers into development teams is beneficial for the developers as well, because an experienced technical writer can provide an another point-of-view on the use cases of the product (by role-playing as a customer), possibly influencing the nuances of the implementation.

The requirements for documentation vary across products, but an example of shared characteristics is a need to have a versioning and rendering system in place. This is enabled by using a versioning system like Git and markup languages, that enable the documents to be typeset in a predictable manner.

The process of writing technical documentation is not straightforward, because, even though there are de-facto industry standards as IBM Style Guide[1], the content has to be written with target audience and software in mind. Nevertheless, there are universal rules that has to be met, including, but not limited to:

- Precision: Documentation has to be precise and without a personal opinion.
- Findability: Documentation tends to inflate over time, writers need to maintain a logical entity hierarchy, categorizing the documentation into separate documents named `Guide` or `Book`. These terms can be used interchangeably.
- Minimalism: Less is more.

This thesis will discuss documentation typical for commercial open source software.

1.1 Semantic Markup Languages

Semantic: of or relating to meaning in language.[2]

"*In computer text processing, a markup language is a system for annotating a document in a way that is syntactically distinguishable from the text.*"[3]

Semantic markup languages provide a mechanism to mark certain parts of a raw text to be typeset differently by the rendering software. This enables writers to write both human and machine readable documentation. The markup adds an additional semantic value to parts of the text. Examples of markup usage include command highlighting, marking headings to be typeset differently, enabling usage of admonitions and code listings. Every semantic markup language has its pros and cons and not every language has a universal use case.[4]

The following part will briefly introduce some of the semantic markup languages: AsciiDoc, DocBook and widely used MarkDown.

AsciiDoc and AsciiDoctor

"*AsciiDoc is a text document format for writing notes, documentation, articles, books, ebooks, slideshows, web pages, man pages and blogs. AsciiDoc files can be translated to many formats including HTML, PDF, EPUB, man page. AsciiDoc is free software and is licenced under the terms of the GNU General Public License version 2 (GPLv2).*"[5]

AsciiDoc is along with DocBook widely used in production, and its importance is backed by support of platforms such as GitHub or GitLab. Simplicity and shallow learning curve of these languages is demonstrated by Listing 1.1 and its AsciiDoctor–rendered version in Figure 1.1. AsciiDoc supports variables and conditional statements, which enable flexibility with the writing. This enables, for example, including a several versions of a documentation that share the same text in a single document and rendering it selectively.

AsciiDoctor[1] is an example of a rendering tool used in production. It has an ability to generate multiple formats from the single source code, for example PDF, HTML, DocBook, LaTeX and ePub. This enables generating content for direct online publication as well as in a downloadable format. AsciiDoctor includes its own "brand" - document presentation style. The presentation style can be customized.

[1] `https://asciidoctor.org/`

Listing 1.1: AsciiDoc syntax demonstration

```
// This is a comment
include::attributes.adoc[]
// AsciiDoc supports including of other .adoc files.

:attribute1: attributes

:toc: // generate an interactive table of contents

= Heading
== Heading
This is a plain-text demonstrating usage of {attribute1}.

IMPORTANT: This is an admonition.

* Simple
* Bullet
* List
```

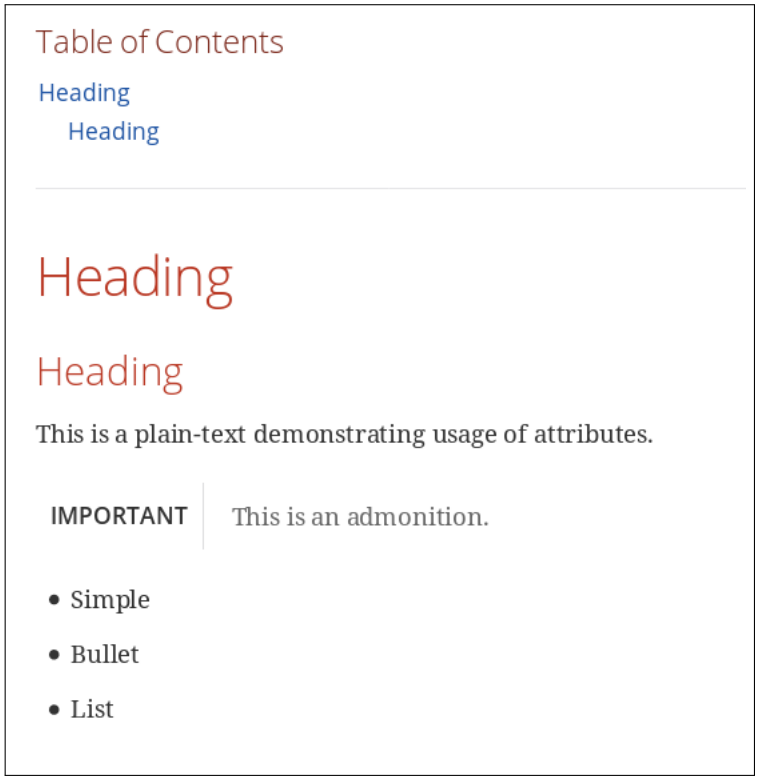

Fig. 1.1: Rendered sample AsciiDoc document

DocBook and Publican

DocBook semantic markup language is derived from XML and is semantically equivalent to AsciiDoc. DocBook uses tagging elements similar to HTML[4].

DocBook elements are divided into three classes:

- **Structural elements** set the type of the document and define the characteristics of internal elements, for example if the text is wrapped in tag `<book>` definition of chapters and headings are possible. Examples of structural elements are `<book>`, `<set>` and `<article>`.
- **Block elements** are similar to HTML block elements. A line break is invoked at the end of every block element. Examples of block elements are listings - `<screen>`, `<itemizedlist>` and paragraph `<para>`.
- **Line elements** are similar to HTML elements as well, they add a semantic value to the text parts, for example marking of software packages `<package>`, e-mail addresses `<email>` and software commands `<command>`[4].

Documentation written in DocBook can be divided into several files, but the files has to be aggregated in one `master` file. DocBook also supports `entities` – parts of code that can be included conditionally or in multiple locations without redundant code. Possible use case is for including a "Product name" or "Product version". Entities save writer from code refactoring when these attributes change [4]. **Publican**[2] is one of the tools for rendering of a DocBook files used in production. It has functions for preparing a required folder structure and, similarly to AsciiDoc, implementing a document presentation style "brand",[4] .

MarkDown

MarkDown[3] is a markup language for writing structured text, most notably in form of *readme* files in open-source projects and is natively supported by versioning platforms, such as GitHub[4] and GitLab[5]. Example of MarkDown syntax is in Listing 1.2.

Listing 1.2: MarkDown syntax demonstration

```
# Heading, level 1                                                    1
## Heading, level 2                                                   2
### Heading, level 3                                                  3
**bold text**                                                         4
- __[GitHub](https://github.com/)__ - GitHub Homepage                 5
```

[2] https://fedoraproject.org/wiki/Publican
[3] https://daringfireball.net/projects/markdown/
[4] https://github.com/
[5] https://about.gitlab.com/

1.2 Automated Testing of Technical Documentation

Ensuring that the technical documentation is correct plays a major role in the process of technical writing. Although, a number of tasks in the process are very repetitive and this can introduce a lot of unnecessary mistakes as a result.

Another part of the documentation testing is a peer review, but it is very time consuming and doesn't provide the same level of reliability as automated testing frameworks.

The factors that has to be tested vary from product to product and also by internal policies, for example IBM Style Guide [1]. IBM Style Guide has become a major influence on technical writing and defines a set of rules that should be met in order to produce a professional documentation.

As semantic markup languages have specific rules that has to be met, and depending on a documentation builder, there are no means to enforce all rules with an universal test. This requires a flexibility in the testing framework and a possibility to omit some of the tests for certain scenarios.

Besides testing of overall documentation integrity and accuracy, technical documentation contains constructs that can be categorized and individually tested, including:

- Hyperlinks,
- Software package names and functionality,
- Blacklisted text constructs or words, for example IBM Style Guide defined phrase/word blacklist,
- Redundant words,
- Typos.

An example how automated documentation testing helps is spell checking. Even though IDEs today contain sophisticated algorithms, these are not perfect and commonly not extensible with an external rule checker (or they are not easily extensible).

Another example would be link testing, where the testing framework can test all hyperlinks individually, process the server response and report it to the user.

1.3 Continuous Integration / Continuous Deployment (CI/CD)

"*Continuous integration is the practice of routinely integrating code changes into the main branch of a repository, and testing the changes, as early and often as possible. Ideally, developers will integrate their code daily, if not multiple times a day.*"[6]

As with software code, documentation set needs to be rendered. Often a rendered documentation contains incorrect format. This can be avoided by providing a preview mechanism before merging the proposed changes to the main code stream. Some servers provide this functionality in form of static page hosting (GitHub Pages) or as a combination of CI/CD and static page hosting (Netlify CI/CD). Netlify CI/CD also provides "pull request previews" that automatically build a documentation preview for each pull request committed in a supported Git hosting server. These previews are made available by sending an automatically generated hyperlink as a pull request message.

The motivation behind CI is to avoid reviewing and testing a large amount of code at the same time. Author's technical writing experience proves that this applies to documentation as well. A paragraph shorter than this one might require feedback from several people in order to be technically accurate. A text change with size of this section can very likely become unmergeable.

CI principles are supported by tools such as issue trackers, for example, JIRA[6] and versioning systems supporting branching. These systems are, on their own, not sufficient and effective CI implementation, writers have to learn and implement habits such as not posting a large amount of text at the same time.

Branching and CI principles have its downsides as well, for example, if the required change is minuscule, the user has to go through the same process (create an issue, create a branch, post a pull request, resolve conflicts, close issue) as with larger changes.

The code has to be verified before it is merged into the production branch, this is where automated testing becomes a part of CI. Tests are ideally executed as soon as the code change is proposed. Automated documentation testing as part of a CI/CD pipeline was a motivation behind creating Emender documentation testing framework.

In terms of Continuous Deployment/Delivery (CD) - a fully automated product

[6] `https://www.atlassian.com/software/jira`

delivery, the current situation is that the final verification (sanity check) of the documentation has to be done manually, so this term does not currently apply in the context of delivery of technical documentation.

Jenkins CI/CD

"*Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.*"[7]

Jenkins is one of the most popular open-source CI/CD platforms, written in Java. It supports variety of version control tools. Typical use case is an automated software builder, where Jenkins is connected to a version control system and any change triggers a predefined set of actions, for example, a build sequence. Its popularity generated a vast variety of verified plugins. Jenkins can be used both from GUI and CLI.

For automation of the technical documentation process, Jenkins is used in similar manner. It builds the documentation set and provides reports about the builds. Implementation of Emender into Jenkins enables the platform to generate test result trends in form of graphs and a "weather report" in Jenkins Dashboard.

Netlify CI/CD

Netlify[7] is a freemium product suite, which goal is to provide a CI/CD platform for web development and deployment. It has a seamless integration with GitHub and can be configured to fetch, build and deploy a website with every code change. It utilizes Linux servers, and can be configured with custom tooling, such as Antora[8]. Users can get the rendered content hosted on the Netlify subdomain or can configure the page to be deployed on self-managed servers with custom domain name. It can utilise GitHub Webhooks[9] and GitHub commit checks, where a hyperlink with rendered documentation preview can be accessed directly from GitHub Pull request.

[7] `https://www.netlify.com/`
[8] `https://antora.org/`
[9] `https://developer.github.com/webhooks/`

2 Emender - Documentation Testing Framework

Emender is an open-source documentation testing framework written in Lua programming language for Linux. It enables creation of Lua test scripts that can be run upon documentation sets written in AsciiDoc or DocBook. If the required documentation structure is implemented, Emender will provide various functions for content analysis, Jenkins integration with REST API and several result presentation methods. Implementation of this framework enables users to focus on test design and not, for example on the parsing methods of the content. Emender has been successfully implemented in production on CI/CD pipelines, testing extensive documentation set for a commercial product.

The Emender tests are run in Jenkins by executing a shell script that converts the documentation into DocBook and runs the Emender framework with a Test Pack written in Lua passed as an argument of a shell script to Emender.

2.1 Emender Framework Structure

Developers of the Emender decided to divide the core into two modules - internal functions and libraries, hosted in **emender**[1] GitHub repository, and external libraries, hosted in **emender-lib**[2] GitHub repository. Libraries from both repositories are required in order to implement Emender.

This section will briefly list the available functions.

Internal Functions and Libraries

Internal functions include, but are not limited to functions for:

- File manipulation,
- Text parsing,
- Table manipulation,
- Presenting the results in various formats, including HTML, XML, JSON and plain text,
- Functions that handle test result printing (with tags `FAIL`, `WARN`, `PASS`, `ERROR`, `INFO` and others).

[1] `https://github.com/emender/emender`

[2] `https://github.com/emender/emender-lib`

External Libraries

The external libraries add support for:

- DocBook markup language,
- XML files,
- Publican, a DocBook documentation builder,
- Several functions for handling of SQLite3 database files.

2.2 Emender Tests

Tests are implemented as Emender, in Lua language and executed by passing them into an `emend` CLI provided by Emender. The tests are structured in the following manner, enabling grouping similar tests into one unit:

Emender Test Structure

- `Test Pack`: Contains a number of `Test Suites`. `Test Pack` is represented by a whole test result file, for example, `results.json` or `results.xml`.
- `Test Suite`: Contains a number of `Test Cases`. Can contain zero tests, if more test granularity is not needed.
- `Test Case`: A singular unit, produces statuses and status messages for a particular test

2.3 Presentation of Test Results

Emender writes test results dynamically to several file types in the folder from which command `emend` is executed:

- JSON in Listing 2.1,
- Plain-text results in Listing 2.2,
- HTML results formatted with Bootstrap, Figure 2.1
- JUnit XML test results consumed by Jenkins CI/CD to generate test result graphs and calculate test result trends, Listing 2.3.

Listing 2.1: Example results.json results file generated from a Test Pack

```
{
  "metadata" : {
      "name":"Guide-Name"
    },
  "results" : {
    "Test_Suite_1": {
      "Test_Case_1": [
        {
          "status":  "fail",
          "message": "The YEAR entity should include 2020.
          Found: '2019' "
        }
      ]
    }
  }
}
```

Listing 2.2: Example plain-text results file generated from a Test Pack

```
---------------------------------
:: CustomerPortalRequirements ::
---------------------------------

Description:     Checks if the guide is prepared
to be published on the Customer Portal.
Authors:         <author_name>
Emails:          <author_email>
Last Modified:   <date>
Tags:            DocBook, WritingStyle
Required tools:

Test Case: testChunkableTagsIDsTag                        PASS

[ INFO ]  Checking
**en-US/Desktop_Migration_and_Administration_Guide.xml**.
[ PASS ]  All **2** chunkable tags are ok
```

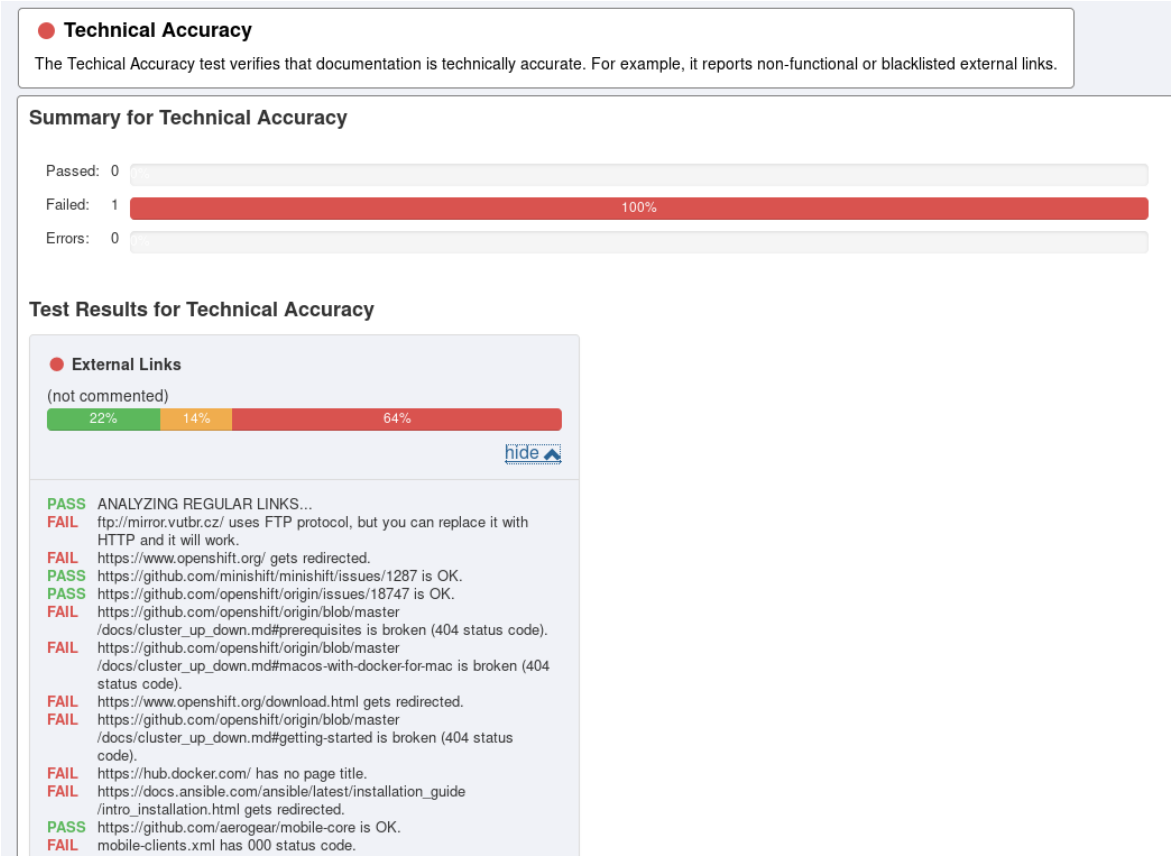

Fig. 2.1: HTML with Test Results, figure is cropped to fit the page.

JUnit XML Test Results

This file is consumed by Jenkins to generate statistics in Jenkins Dashboard, and mirrors the structure of the JSON file in a specific XML format. JUnit XML files reports only failed tests, non-reported tests are considered passed.

Listing 2.3: Example JUnit XML file with Test results

```
<testsuites>                                                    1
  <testsuite name="CustomerPortalRequirements">                 2
    <testcase name="testCase1" classname="testCase1">           3
    </testcase>                                                 4
    <testcase name="test1" classname="test1">                   5
      <error message="Test failed.'">Test failed.</error>       6
    </testcase>                                                 7
    <testcase name="test2" classname="test2">                   8
    </testcase>                                                 9
  </testsuite>                                                  10
  <testsuite name="GitMergeLeftovers">                          11
  </testsuite>                                                  12
</testsuites>                                                   13
```

2.4 Implementation of Emender on Documentation

This section will discuss installation and implementation of Emender onto a documentation set. Emender documentation for installation and implementation is incomplete, so the implementation of the existing *Test Suite* was problematic. This section will serve as a guide how to install Emender and implement "Technical Accuracy Tests"[3] onto a documentation set.

Installing Emender

Emender is hosted on GitHub and its core consists of two repositories – `emender`[4] and `emender-lib`[5]. Resources from both repositories have to be installed with instructions present in the respective `README.md` file. Each repository contains a `Makefile` that facilitates the installation.

[3] `https://github.com/emender/technical-accuracy-tests`

[4] `https://github.com/emender/emender`

[5] `https://github.com/emender/emender-lib`

Requirements for the document and document structure:

- Tested AsciiDoc file has to be named `master.adoc` ("include" AsciiDoc directives can be used in this file). A workaround for using different names is to edit the shell script `run.sh`, but this is generally not recommended.
- It is necessary that text in AsciiDoc does not contain "less-than" (`<`), or "more-than" (`>`) characters. This causes the test to run and seemingly exit without errors, but the test fails to detect any links. This is because Publican fails to build the documentation from DocBook format that is converted from AsciiDoc. "Less-than" and "more-than" signs are reserved characters in XML format, and if not escaped properly, will cause Publican to fail. Escaped characters can be used instead.
- Shell script `run.sh` has to be executed from the documentation folder, or anywhere with `-XtestDir=<docDir>` referring to the documentation folder.

The shell script doesn't have error handling, so these requirements had to be tested manually.

Procedure executed by the testing shell script

If the requirements for the document and document structure are met, the shell script will execute the following procedure:

- Recognise the input file format (AsciiDoc, DocBook or other)
- If script detects an AsciiDoc format, it creates a DocBook project and converts the AsciiDoc into DocBook. If the conversion fails, it wont report an error, but the Test Pack will subsequently report no hyperlinks found.
- Run the Test Pack and write the results into XML, JUnit, HTML, JSON, and plain-text files.

Executing Emender tests

Tests are implemented by writing tests in Lua, utilising Emender libraries. The tests are executed with command **emend** and proper arguments as per **emend** man page. Tests natively support DocBook language. AsciiDoc can be used, but the document has to be converted to DocBook prior running the tests. In the Test Packs available in **emender** GitHub project, this is done with **asciidoctor** and **publican** by recognizing the file structure, and if needed, converting into DocBook format and subsequently running the Test Pack, all within a single Bash script[6].

[6] `https://github.com/emender/technical-accuracy-tests/blob/master/run.sh`

2.5 Examples of Implemented Test Suites

GitMergeLeftovers

Verify that the book does not contain any Git Merge Leftovers. Git merge procedure uses strings of less-than(<), more-than(>) and equal (=) signs to differentiate between existing and incoming changes. These are often left in the documentation.

TestPackages

Test that the documentation contains only current versions of the packages. This Test Suite apply only for DocBook as AsciiDoc does not contain a markup for packages.

- `testCommandTag` - Test packages discovered from parsing the command in a tag `<command>`.
- `testPackageTag` - Test packages discovered from parsing the command in a tag `<package>`.

GuideStatistics

Collects various statistics about the book, for example, page count, word count, number of used graphics, frequency of used tags, and others.

TestLinks

Test suite verifies that all external links are functional, for example:

- The test reports codes 403, 404 or 500 (and fails),
- FTP links, if HTTP protocol is also available but not used then fail,
- Red Hat customer portal links (access.redhat.com),
- Blacklisted hyperlinks,
- Redirected hyperlinks (test fails),
- Hyperlinks without page titles (test fails).

TestWritingStyle

This Test Suite for DocBook and partly with AsciiDoc . Tests violations in writing style. The Test Suite can use external dictionaries that serve as blacklists or whitelists. Examples of Test Cases:

- `testSpellChecking` compares documentation external blacklists and whitelists.
- `testSentenceCase` Tests that the sentences have capital letters only in the beginning of the sentence.

3 False Positives in Tests

A false positive is demonstrated on example from JSON file holding test results generated by test from Emender *Test Suite* "Technical Accuracy Tests". The test is testing hyperlinks and reporting a test failure after detecting a FTP URL. The result is not necessarily "false", but it holds an informative value to the user.

Listing 3.1: Truncated results.json with false-positive result

```
{
  "metadata" : {
    "name":"unknown"
  },
  "results" : {
    "TechnicalAccuracy": {
      "testExternalLinks": [
        {
          "status":  "fail",
          "message": "ftp://mirror.vutbr.cz/
          uses FTP protocol,  but you can
          replace it with HTTP and it will work."
        }
      ]
    }
  }
}
```

False positives can be induced in tests deliberately or can result from imperfect test design. The possible motivation behind deliberate induction of false positives might be to bring the attention to a certain part of a tested documentation, even if it is grammatically and syntactically correct. Possible real-life scenario might be an enforcement of internal writing style rules or as 3.1 shows, to discourage using of FTP protocol when HTTP protocol is available. Currently, Emender does not have a functionality to handle false positives.

4 Current Implementation of Emender in Jenkins CI/CD

This section describes the current implementation of Emender tests in Jenkins CI/CD infrastructure used by company's documentation team. Even though the framework can be run locally, this configuration automates the build and testing process and ensures access to the results to all technical writers in the company.

4.1 Characteristics of the Implementation

- The documentation set is hosted in a Git repository.
- Emender test framework is installed on the same machine that hosts Jenkins CI/CD.
- Each book (guide) has a specific job (item) created in a Jenkins that is connected to the specific Git repository.
- The Jenkins job (item) is configured to run a specific *TestPack* on a documentation set, triggered by a Git commit.
- The results are generated as build artifacts in Jenkins. (static HTML, JSON, JUnit XML and plain-text results.)
- The test results are processed by Jenkins with a JUnit plugin, consuming JUnit XMLs that Emender creates as a build artifact. [1]

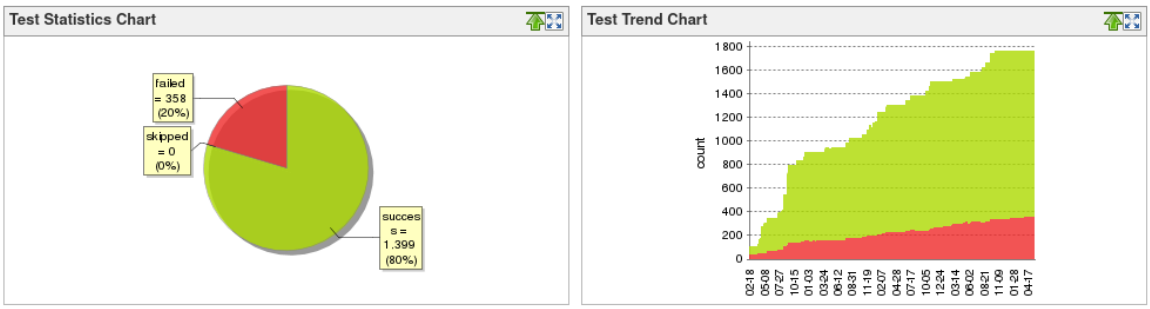

Fig. 4.1: Test result trends generated by JUnit plugin in Jenkins.

Jenkins Dashboard provides a build "weather report", where user can see trends in test results – if the tests are failing frequently (in multiple succeeding builds) the "weather" is reported as sunny, cloudy or stormy, respectively.

[1] https://plugins.jenkins.io/junit/

4.2 Viewing Emender Generated Test Results

If the configuration is correct and the Test Pack was executed successfully, Emender generates several test result formats, one of them being a structured JSON.

The structure of the result files reflect the three-level structure mentioned in 2.2

4.3 Drawbacks of the Current Implementation

Bad accessibility of test results

Main drawback of a Emender is that it does not provide a way to comfortably work with the results for an inexperienced user. It was developed primarily for Jenkins CI/CD deployment and depends on a user's working knowledge of Jenkins and it's configuration to run Emender in order to be able to see and work with the test results:

- The tests are available only through Jenkins Dashboard, which is also used for rendering of the books – Jenkins instance is hosting a large number of projects.
- The Jenkins host machine is running a significant number of jobs and that makes connection to it very slow.
- The tests are not available by their commit ID but by a job number that increments with each commit, which makes finding a specific Git commit difficult.
- The results in HTML are only readable when user downloads the entire job artifact folder from Jenkins (the HTML is dependent on Bootstrap and jQuery).

No mechanism for false positives handling

Another limitation of Emender documentation framework is that tests regularly report false positives (tests failing on documentation that is written correctly). This can be solved by implementation of a suitable "waiver" module.

The design of a web application presented in the following sections tries to overcome these shortcomings and provide the user with a graphical interface that can be accessed via a concise REST API.

5 Proposed Improvements for Emender

A proposed improvement for Emender implementation in CI/CD is to design an independent web application *emenderwebservice*. The application will serve a dynamic test result website with a mechanism that allows waiving of false positive results. The motivation behind a service independent from Emender is to avoid rewriting the Emender core and subsequently the existing tests. The web application will be subsequently incorporated into CI/CD pipeline for complete test automation.

The following section will list the requirements that were identified for the implementation. These will be described in more depth in later sections.

5.1 List of Requirements

- Storing the Emender test results with unique identification in a database,
- A graphical user interface to present the test results,
- A mechanism to handle false positive results,
- A JUnit XML result file generator for each test result pack,
- REST API endpoints for:
  - Retrieving the test results from Jenkins CI/CD,
  - Returning results in JUnit XML format to Jenkins CI/CD,
  - A graphical user interface.

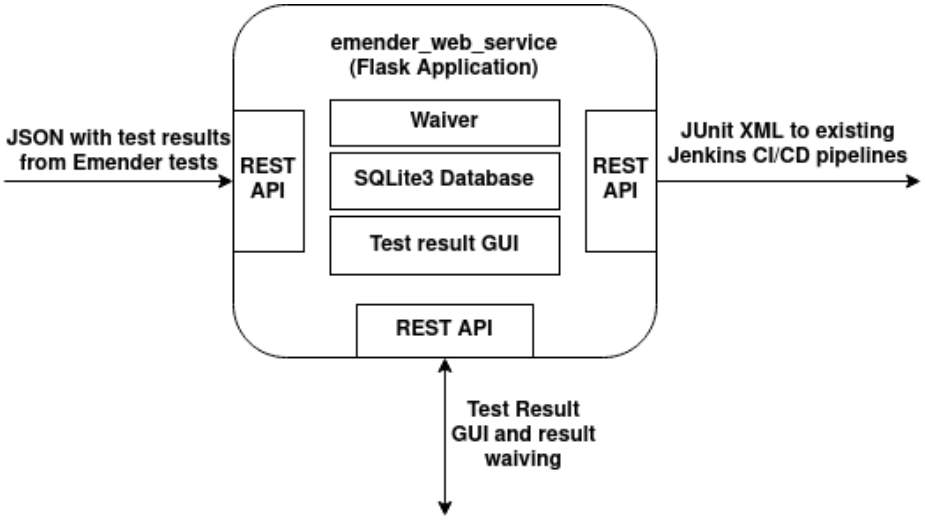

Fig. 5.1: emenderwebservice design diagram

5.2 Identification of Emender Test Results

In order to properly store, display and manipulate the test results, their proper identification has to be secured. JSON result files generated by Emender in Jenkins CI/CD only contain structured test results (see Section 2.2), and limited information about the tested document and the parent documentation set.

The identification of test results is divided into two sections – unique identification of a specific revision (Git commit) of a guide and unique identification of test results within the JSON results file.

5.2.1 Attributes Used for Identification of Documentation and Test Results

Attributes uniquely identifying a specific guide (book):

- **Product Name** (*OpenShift*),
- **Product Version** (*Dedicated, Container Platform*),
- **Book Name** *(Getting Started, Migration Guide)*,
- **Git Commit ID** identifying a specific Book revision.

Attributes uniquely identifying test results. These attributes reflect the test structure in Section 2.2:

- **Test Pack** name is a collection of tests run by Jenkins that produces a single `results.json` file,
- **Test Suite** is a collection of Test Cases,
- **Test Case** is a most granular element in this collection, however, it runs a number of tests under one **Test Case**.

5.2.2 Uniquely Identifying a Specific Book

A specific book is identified by a unique combination of **Product Name**, **Product Version** and **Guide Name**.

5.2.3 Uniquely Identifying a Specific Git Revision of a Book

A specific Git revision of a book is identified by a unique combination of **Product Name**, **Product Version**, **Guide Name** and **Git Commit ID**

5.2.4 Uniquely Identifying JSON Files with Test Results

A JSON test result file is associated with a specific Git revision of a Book and a specific Test Pack that was executed on the Book. The JSON file only contains information about Test Pack and Tests executed within it, the information about a Book has to be received from another source.

A method for acquiring the missing information is to create a unique REST resource for each **Product**, **Product Version**, **Guide** and **Test Pack** (combined):

```
/api/results/<product>/<version>/<guide>/<git_commit>/<test_pack>
```

This endpoint will serve as an endpoint for Jenkins CI/CD generated test results (in JSON format). A Jenkins build for a specific **Guide** will have to be configured manually to send the test results to a specific endpoint identifying this **Guide** and a **Test Pack** that was executed on this book. For example, `curl`[1] can be used to send the results to *emenderwebservice*.

For brevity, the following text will use an abbreviation `<test_results>` as an abbreviation for `<product>/<version>/<guide>/<git_commit>/<test_pack>` as it represents a single JSON file.

5.2.5 Uniquely Identifying the Results within the JSON File

The information gathered in Section 5.2.4 uniquely identifies the JSON generated by the Emender run by the Jenkins CI/CD. In order to handle the test results and to enable an implementation of a waiving mechanism, the identification has to include more granularity.

These attributes have been identified as necessary for unique identification of test results:

- **Test Suite** name (for example: TestLinks),
- **Test Case** name (for example: FTP link check, 404 checks),
- **Results** from **Test Cases**: **Test Cases** are not granular and hold several tests under one name.

These two sets of attributes will be categorized in a SQLite database tables with an appropriate relationships between tables.

[1] `https://curl.haxx.se/`

5.3 Storing the Test Results in a Database

In order to make the results widely available, a reliable storing mechanism has to be implemented. The test results will be stored in a database that enables their unique identification and easy access.

SQLite3[2] RDBMS (Relational Database Management System) has been chosen for its widespread use, flexibility, simplicity and support by Flask framework and Python.

5.4 Graphical User Interface (GUI) with a Waiving Mechanism

GUI

A graphical user interface that will dynamically present the test results in a web page. The web page will source the information from the SQLite3 database. The GUI will implement a mechanism to handle false positive results ("waiver").

Mechanism to handle false positive results

Waiver is used to mark false positives in test results. With complete test automation, when tests are implemented as a part of CI/CD, a falsely failed test will cause CI/CD pipeline to fail as well. Tests can be (and some of the tests are) designed so certain results are marked as `FAIL` even if the documentation is valid, to get the user's attention to the content, as stated in Section 3. Waiver gives user an option to disregard these results and therefore further customize the testing infrastructure.

Proposed implementation is a dynamic generation of HTML input elements in a GUI that will register the waived failed result in a database with the test results.

5.5 JUnit XML Generator

The existing implementation generates test result trends by a JUnit plugin in Jenkins CI/CD. Because the results will be altered with the waiving mechanism, a new JUnit XML will have to be generated and provided in a REST API for each commit(build), so Jenkins can be configured to download and process the file. This makes the implementation compatible with the existing infrastructure.

[2] `https://www.sqlite.org/index.html`

6 emenderwebservice in Jenkins CI/CD

This section will introduce the process of generating the Emender test results with Jenkins and how emenderwebservice fits into it.

The process requires the Jenkins CI/CD to be configured to run the Test Pack on a specific guide stored in a Git repository, send the JSON test results to emenderwebservice and subsequently retrieve the JUnit XML file from the application's REST API.

This section will briefly introduce the standard use case scenario of emenderwebservice with Jenkins CI/CD.

Generation of Emender test results by Jenkins

When a user makes a Git commit to a documentation repository specified in a Jenkins job, Jenkins runs a certain **Test Pack** on the documentation. This job generates several result files, one of them being `results.json`.

Sending the results to a emenderwebservice REST API

Jenkins CI/CD sends `results.json` to emenderwebservice application's REST API to a specific URI: `/api/results/<test_results>` where `<test_results>` represent:
`<product>/<version>/<guide>/<git_commit>/<test_pack>`. This URI uniquely identifies the results, and serves as an identifier for a web application on how to store the results.

JSON validation with JSON Schema

The emenderwebservice validates the JSON with a JSON schema and if the JSON is valid, it is handled for a further processing. If the JSON is invalid, the REST API responds with an error message in JSON format and a HTTP error code.

Generation of an Test Results HTML with a "waiver" functionality

The server generates the HTML with a waiving functionality. It means that if the test is reported as `FAIL` a checkbox is generated next to the result that enables user to "waive" the failed result as a false positive. This is handled by an AJAX jQuery JavaScript and sent to a emenderwebservice REST API for storing the waived result in the database.

The server generates an HTML and serves it on the same URI as where the results were sent: `/api/results/<test_results>`. The list of all available test results pages will be available at the root URI `(/)`.

emenderwebservice generates JUnit XML for Jenkins test result trends

The server generates a JUnit XML file with a list of Test Suites and Test Cases, reporting failed tests. This JUnit XML will be available at a similar URI as test results: `/api/junit/<test_results>`. The JUnit XML file is processed by the Jenkins JUnit XML plugin used to generate test reports in Jenkins dashboard.

The next chapter will describe the technologies that will be used to implement the solutions.

7 Technologies Used For Implementation

This section will briefly introduce the technologies and standards used for the implementation of emenderwebservice.

7.1 REST and RESTful Web Services

"*Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating web services. Web services that conform to the REST architectural style, termed RESTful web services, provide interoperability between computer systems on the Internet.*"[8]

- REST was designed to provide a uniform API between web services. It has become de-facto a standard for communication between web services.
- REST provides uniform and predefined set of stateless operations, meaning that the communicating parties have no prior knowledge of the state of the other.
- In a RESTful web service, requests made to a resource's URI will elicit a response with a payload formatted in either HTML, XML, JSON or other format. The response can confirm that some alteration has been made to the stored resource.

REST Terminology

REST uses terminology defined by World Wide Web Consortium (W3C)[8]:

Representation: "*A representation is a piece of data that describes a resource state.*"

Web service: "*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.*" Web services communicate by exchanging messages.

Resource: "*A resource is defined to be anything that can have an identifier. Although resources in general can be anything, this architecture is only concerned with those resources that are relevant to Web services and therefore have some additional characteristics. In particular, they incorporate the concepts of ownership and control.*"

Identifier: "*An identifier is an unambiguous name for a resource.*" An identifier should be realized with a URI. An identifier identifies a resource that is relevant to the architecture.

URI/Resource Relationships: "*By design a URI identifies one resource.*" **Resource** is used for anything that might be identified by a **URI**. All of resource's essential characteristics can be conveyed in a message.

Idempotency: Being idempotent in means that a specific request will always elicit the same response. For example if client requests server to update a resource with a HTTP method `PUT`, it will overwrite the existing resource present on the particular URI. In contrast, HTTP method `POST` does not elicit the same response, sending a request with a HTTP method `POST` will always create a new unique resource.

REST, CRUD and HTTP Methods

REST, being a specification, not a protocol, does not define a specific HTTP methods (or, in some literature, *HTTP verbs*)[9] that should be associated with either Create, Read, Update or Delete functions.

Even though that are no recommendations for using a specific HTTP verb for a specific CRUD function, a chosen implementation should be consistent across modules in a given application [10], [11].

HTTP verbs are used to identify what operation client wants to do on a particular resource, REST specification uses those to identify the CRUD methods. HTTP verbs are transmitted in a header of HTTP request.

REST and JavaScript Object Notation (JSON)

As stated in the previous section, REST is a specification, not a protocol, the format of response objects (send in the body of a HTTP response) is also flexible, but a recommendation is to use either JSON or XML format. The application developed in this thesis uses responses in JSON format[12].

7.2 Flask Web Development Framework

Flask[1] web application framework has been selected for the implementation for its simplicity and for being Python native. Flask is a popular web microframework. "Micro", means that it requires little to zero boilerplate code in order to implement a complete web application. It has a built-in development server that enables quick deployment and debugging. Flask is by its nature extensible by plugins (implemented as Python modules).[13]

Examples of companies that use Flask for web development include Netflix, Red Hat, Reddit, Airbnb.[14]

Flask was designed to be a modular framework, by itself providing a limited functionality. For example, it does not provide any database support, but relies on a number of extensions (for example, Python libraries) to provide the functionality.

Flask main Python dependencies:

- Werkzeug, providing WSGI and routing and debugging subsystems
- Jinja2, a templating engine
- Click, CLI integration

The following chapter will focus on introducing the Flask dependencies Werkzeug and Jinja.

Werkzeug Web Server Gateway Interface (WSGI)

"*Web Server Gateway Interface is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request. WSGI is a Python standard described in PEP 3333.*"[15] [16]

WSGI is an API that allows web servers to communicate with Python web applications. It is an interface specification, not a framework or a library.[17]

A WSGI server serves as a "translator" for request/response pairs exchanged by the web server and a Python application. An example of WSGI server is Gunicorn[2].

[1] `http://flask.pocoo.org/`

[2] `https://gunicorn.org/`

Werkzeug HTTP request and response processing

WSGI provides full HTTP request processing capability, where the outgoing data are constructed to be a valid HTTP response and the incoming HTTP requests are directly parsed into Python objects.

In Flask, the HTTP request/response pairs are handled by Python functions and objects. Before this is enabled a Python module has to be "registered" as a Flask application by creating a *Flask* object with the Python module passed as an argument.

Route Handling

"*The association between a URL and the function that handles it is called a route.*"[18]

Listing 7.1: Example Flask route

```
@<app_name>.route('/api/', methods=['GET'])
def example():
  return '<h1> Example </h1>'
```

The route handling is implemented by using a Python function decorator `@app.route` that parses and provides the URI segments to the function it decorates.

The Listing 7.1 demonstrates a handling of `/api/` route, where the route handles a `GET` request send to URI `<serverip>/api/`. The function creates a HTTP response with a simple HTTP heading and sends it to the client.

`app` used in the function decorator is a Flask object that was initialized in the application definition file.

Flask class `Response` is used for response construction and can have several forms, for example a rendered Jinja template with or without passed variables, JSON response constructed by a `jsonify()` function or a simple HTML code as showed in the Listing 7.1. The `Response` object is created as a return value of the `route()` function.

The default media type `mimetype` of the response is `text/html`, but this can be set according to the required response type, for example `jsonify` sets the `mimetype` to `application/json`.

Flask folder structure

Flask recognizes the project files in a specific folder structure:

- `/templates` for Jinja template files,
- `/static` for static structures such as JavaScript code, Bootstrap source files and images,
- `routes.py` for route definitions.

7.3 Jinja Template Engine

Jinja is a template engine used with Python. Flask utilizes Jinja as a web template engine and the following text will talk exclusively about web templating. Jinja templating is also implemented in an automation tool Ansible. [3]

"*A template engine is software designed to combine templates with a data model to produce result documents.*" [19]

Template is a text-file containing constructs that enable template engines to dynamically generate web pages (or, in general use – any source code).

This functionality is utilized in this thesis to create a dynamic web site that handles and provides a GUI for Emender documentation test results.

Jinja Templates

A Jinja template, rendered by a Python function `flask.render_template()`, provides a connection between a Python application and the HTML response with access to variables passed from a Python script. It integrates Jinja functions, providing, among others looping functions and conditional constructs. These constructs can be used to dynamically generate a web site content.

In Flask, any code can be incorporated and handled by a Jinja template. An example shows how a simple iteration over a list passed by a Python script can generate an HTML document. To maintain separation of concerns, all relevant logic should be isolated from the template, as much as possible. To avoid injection attacks, *MarkupSafe* Python package is installed as Jinja dependency. *MarkupSafe* escapes untrusted input when rendering templates.[4]

[3] `https://www.ansible.com/`

[4] `https://markupsafe.palletsprojects.com/en/1.1.x/`

Listing 7.2: Example Jinja HTML template generating a simple URL list

```
{% for URI in uri_list %}
  <a href="{{ URI }}">{{ URI }}</a><br/>
{% endfor %}
```

Listing 7.3: Example HTML generated with a Jinja template

```
<a href="http://www.google.com">http://www.google.com</a><br
   />
<a href="http://www.vutbr.cz">http://www.vutbr.cz</a><br/>
<a href="https://flask.palletsprojects.com">https://flask.
   palletsprojects.com</a><br/>
```

7.4 SQLite3

SQLite3[5] is an SQL relational database that requires zero configuration, is serverless (It does not require a server. the database is stored in a local binary file) and fully supported by Python standard library *sqlite3*. Because of these lightweight characteristics, the library was chosen as the database engine for this project.

7.5 Converting Python Applications to Web Applications with Flask

Flask, along with Werkzeug and Jinja, enables developing and connecting of Python application so the developer can integrate Python libraries for variety of tasks and does not have to rely on the plugins developed natively for the framework. In other words, a Python application that was originally designed to output information to a console or a local graphical user interface without networking capability can be easily, with Flask, transformed into a Web application and deployed on a WSGI server.

[5] `https://www.sqlite.org/about.html`

8 emenderwebservice Implementation

This chapter will describe the implementation of `emenderwebservice` to requirements defined in Chapter 5 using technologies specified in Chapter 7.

During the design of the implementation there were a number of unsuccessful attempts with technologies that were proven to be inefficient, or the learning curve was very steep. A direct approach was tested to be the most effective, without ORMs (Object Relational Models) or Flask plugins.

The main blocks of the application are:

- REST API management utilizing Flask *routes.py* script,
- Database of resources (test results) implemented in SQLite3,
- Database controller,
- GUI with waiving mechanism generated by Flask templates and JavaScript,
- Validation of incoming JSONs with JSON schema,
- JUnit XML generator.

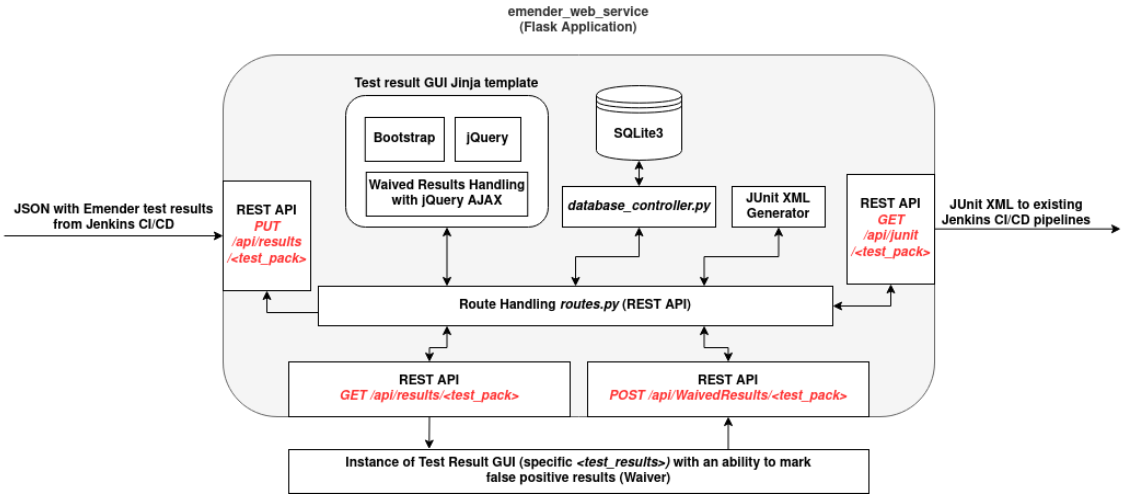

Fig. 8.1: emenderwebservice block diagram

The application was implemented in Flask using software pattern similar to Model-View-Controller. This pattern emphasizes the separation of modules, so these can be developed and tested individually. This method is called a "separation of concerns".

8.1 Model-View-Controller (MVC) Architecture

The implementation tries to follow the principles of MVC, where Models, Views and Controllers are implemented as individual components, thus helping to achieve separation of concerns. With an increasing separation of the components increases the flexibility of the design and an ability to test the components individually.

"*User requests are routed to a **Controller** which is responsible for working with the **Model** to perform user actions and/or retrieve results of queries. The **Controller** chooses the **View** to display to the user, and provides it with any Model data it requires.*"[20]

- **View** in this implementation is represented by Jinja HTML templates.
- **Controller** is represented by a route handling script `routes.py`
- **Model** is represented by SQLite3 database and the database controller, which defines the form of the data presented to a Jinja HTML template.

8.2 REST API Implementation

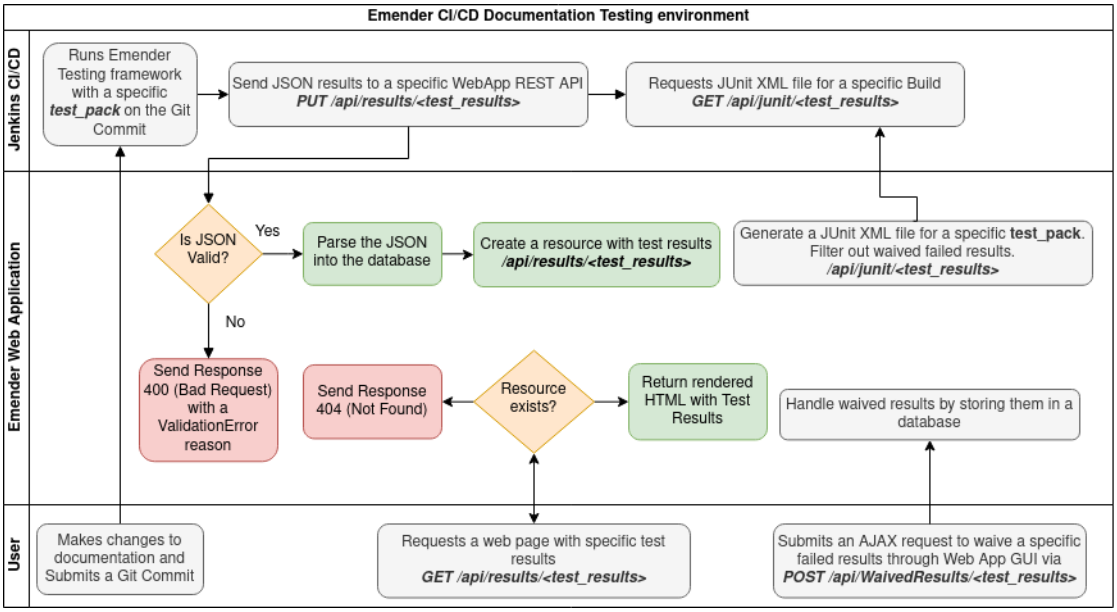

Fig. 8.2: Diagram with processes in Emender web application.

REST API serves as an access point to the application from the network. As stated in Section 7.2, the application is accessible by sending requests to REST API endpoints, which Werkzeug WSGI parses as various Python Flask objects, notably

`Flask.request`, that enables to access various parts of a HTTP request. Full list of available properties are in the Flask documentation [1]

The REST API in emenderwebservice is implemented and the processes managed by the Python script `routes.py`. `routes.py` serves as a controller that constructs views by manipulating data in the database and presenting them through the Jinja templates.

Handling Resources

The resources are created by storing the information in a database. There are plugins for Flask that are designed to help with development of REST APIs, for example *Flask-Restful*, but after experiments, a direct approach was chosen without Python object abstraction.

When the client sends a request for a non existing resource, Flask automatically returns a HTTP `404 Not Found` Status code. Similarly, if the request contains an unsupported HTTP Verb for a given route, an error `400 Bad request` is returned automatically.

8.2.1 REST API Endpoints

This section will describe the implemented REST API endpoints (routes) and the Python functions invoked with the request:

`/api/results/<test_pack>`

A request to this endpoint invokes `rest_test_results()` function, and depending on the HTTP method used, the following scenarios are executed:

PUT method

The PUT method is used, because an idempotent result is anticipated - an existing JSON on a specific URI will be rewritten by the subsequent calls on the same URI, as stated in Section 7.1.

The script parses the values from the URI path variables and validates the received body of the request.

If the received body does not comply with the JSON Schema 8.3, the script returns a `ValidationError` exception in JSON format and HTTP `400 Bad Request` status code in the response.

[1] `https://flask.palletsprojects.com/en/1.1.x/api/#flask.Request`

If the JSON complies with the JSON Schema, the JSON with the variables parsed from the URI path are passed to the Database controller to be stored in the database.

If there is a problem with parsing the values into the database, the Controller sends a JSON status with a HTTP `500 Internal Server Error` status code to the client.

Finally, if the Test results were stored in the database successfully, the script returns a JSON with HTTP `200 OK` status code in the response to the client.

GET method

The script validates the request properties (these are parsed from URI), and sends a request to a Database controller to retrieve the test results. If the Database controller reports that the test results could not be found in the database, the script returns a JSON status with a HTTP `404 Not Found` status code to the client. Otherwise, a script returns a rendered Jinja template `results.html` containing GUI with the test results.

`/api/junit/<test_pack>`

A request to this endpoint invokes `rest_junit()` function and following scenarios are executed:

GET method

The script parses the path from URI and queries the Database controller for results for a specific Book tested by a specific Test Pack. If the Database controller is not able to find the result an error response in JSON and HTTP `404 Not Found` status code is returned back to the client.

Otherwise, a script queries Database connector for a list of waived results for a specific Book and a Test Pack. The list of waived results can be empty.

Finally, the function `junit_rendering()` is called with the results retrieved from the database and the generated response is returned to the client.

`/api/WaivedResults/<test_pack>`

A request to this endpoint invokes `rest_waiving()` function, and the following scenarios are executed:

POST method

The function parses the JSON sent by the AJAX function within a Test Result GUI and requests Database controller to process the waived failed result.

The function returns JSON status messages with HTTP status codes - `500 Internal Server Error` if the Database controller was unable to save the result, or `200 OK` otherwise.

`/api/results` and `/`

A request to these endpoints invokes `rest_list_of_results()` function. The function parses the URI elements and queries the Database controller for a list of available Test Result pages in URL forms. This list is rendered and returned as a HTML document with a Jinja template `list_of_results.html`. `/api/results` redirects to `/`.

8.3 JSON Schema

JSON generated by Emender has a specific structure, illustrated in Listing 2.1. To ensure that only valid JSONs are further processed, a JSON Schema validation is implemented. [2].

Even though JSON Schema is not standardized by a authority, it is widely used and several implementations exists. This application uses the `jsonschema` Python package available from the Python standard library (PyPi).

JSON Schema in 8.1 is presented in a nested JSON format as well, with the data structures describing the required structure.

The JSON schema is validated by `jsonschema.validate()` function and raises an `jsonschema.ValidationError` exception when the tested JSON does not comply to the schema.

Listing 8.1: JSON Schema for Emender test results, truncated

```
{
   "$schema": "http://json-schema.org/schema#",
   "$id": "emender-results-json-schema",
   "title": "Emender Results JSON Schema",

   "description": "JSON Schema for Emender",
   "type": "object",
   "properties": {
      "metadata": {
      "description": "Holds metadata for the results,
         typically a name object.",
      "type": "object",
      "properties": {
        "name": {
        "description": "Holds a name of the guide.",
        "type": "string"
         }
        }, "required": ["name"]
      },
      "results": {},
   "required": ["metadata", "results"]
}
```

[2] `https://json-schema.org/`

8.4 Database Implementation

The database is the core of the emenderwebservice implementation. It is designed to provide a structure for the tests generated by Emender CI/CD that enable a unique identification of tests and their relationship with a specific documentation. The database is implemented in SQLite3, the schema visualizing relationships between tables is included as Appendix A. There were several versions of the database design tested, most notably an attempt in making the test results as granular as possible (by using tables `TestSuite` and `TestResults`). These tables were unused in the final implementation but are left in the design for further possible utilization.

This section will introduce the relationships in more detail and describe the functionality the relationships enable.

The database is implemented in SQLite3 dialect of SQL. There were experiments with ORMs (Object Relational Models) such as SQLAlchemy[3], but the required effort for the steep learning curve was not effective for the implementation.

The SQL script used to generate the database is available in the archive attached with the thesis (`database-definition.sqlite`). The database was created and debugged using *DB Browser for SQLite*[4].

Foreign keys

The relationships between tables in the implementation are defined by foreign keys. To facilitate a unique identification of a row in a table, a Primary key is defined. For a unique identification, the values that are used as Primary key must be unique. This can be implemented by adding a new column with auto-indexing (auto-increment) ability. To define a one-to-many relationship between two tables, this Primary key is defined in the "many" side as a foreign key. This is used extensively in the implementation, as the structure requires cascaded one-to-many relationship to uniquely identify resources.

Constraints

"*Constraints in SQL are used to specify rules for the data in the table.*"[21]

Examples of commonly used constraints include `PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL` and `UNIQUE`. This keywords can define relationships between rows in a

[3] `https://www.sqlalchemy.org/`
[4] `https://sqlitebrowser.org/`

table, for example, to enforce unique combination of values in the rows. This is utilized extensively in the implementation.

ProductName, ProductVersion, BookName tables

This triad is used to uniquely identify a specific Book for a specific `ProductName` and `ProductVersion`. The tables are connected in a cascade of one-to-many relationships with constraints ensuring a unique combination of `ProductName`, `ProductVersion` and `BookName`, for example *RHEL 8 Installation Guide*.

ProductName

Defines a Product Name, has to be unique in the table, for example *OpenShift*.

ProductVersion

Defines a Product Version of a specific product. The value don't have to be unique in the table, but a constraint is implemented to ensure a unique `ProductName` and `ProductVersion` combination.

BookName

Defines a specific Book type, for example *Installation Guide*. The value don't have to be unique in the table, but a constraint is implemented to ensure a unique `ProductVersion` and `BookName` combination, and the cascaded relationship between `ProductVersion` and `ProductName` ensures that there will be unique combinations of `ProductName`, `ProductVersion` and `BookName`.

GitCommitID table

Identifies a specific revision of a specific book. Shares a one-to-many relationship with `BookName` table.

TestPack table

Identifies a `TestPack`. `TestPack` is a highest structure of Tests, defined in Section 2.2.

WaivedFailedResults table

Holds unique identification for waived failed test results. The values are tied to a specific `BookName` and a specific `TestPack`. It is Git-Book-revision agnostic, meaning that the waived results will be valid for all revisions of a `BookName`.

Originally, the design aimed to store the waived test results in `TestResults` table, but because of the table being connected to a specific `GitCommit`, not a `BookName`, this approach was chosen. The table holds values retrieved from the REST API `/api/WaivedResults/`, uniquely identifying the test results with the following columns:

- `TestSuiteName`: A `Test Suite` name
- `TestName`: A Test Case name
- `ResultMessage`: A message reported with the result.
- `WaivedFailedResultUniqueName` is a concatenation of the previous values in the following format `TestSuiteName_TestName[Message]`. This column simplifies the algorithm for rendering of test result template.

This table is used in conjunction with `RawJSONs` table to generate the JUnit XML file and to correctly generate the test result Jinja template (graph calculations and setting the values of waiving checkboxes).

RawJSONs table

Holds raw valid JSON test result files retrieved from the Jenkins CI/CD in a string format. These are used to simplify rendering of test result template. `ResourceURI`s are stored here to simplify the process of rendering the list of available test results with `list_of_results.html` template. The values in rows have to create a unique combination. Used in conjunction with `WaivedFailedResults` to generate the JUnit XML file and to correctly generate the test result Jinja template (graph calculations and setting the values of waiving checkboxes).

TestSuite, TestName and TestResults tables

These tables enable unique identification of tests within a single JSON file. The cascaded relationship structure is identical to that between `ProductName`, `ProductVersion` and `BookName` tables, so it won't be reiterated in the text. `TestResults` table holds uniquely identifiable test results, because it holds a Foreign key of `GitCommitID`. This ensures the row values can be traced to a specific Git revision of a specific book, tested with a specific `TestPack`.

These tables are populated by parsing the JSON file with the database controller. This triad was used in the previous iterations of the implementation design. It can be deleted but is left in the implementation to enable possible further improvements. Downside of this decision is that the database storage will be utilized ineffectively.

8.5 Database Controller

The database controller is a script that utilizes the `sqlite3` Python standard library. The controller uses SQL statements to read and modify the database content. The `sqlite3` module also provides exceptions that can be handled by the script.

Parsing and storing the test results from a JSON file

The function `add_and_parse_json()` processes information retrieved from the client, containing identification of a specific tested book and a JSON file with test results.

The information about the book is processed first by sequentially adding the `ProductName`, `ProductVersion`, `BookName` and `GitcommitID`. Subsequently, the JSON is added in its raw (string) form with a `ResourceURI` containing the URI of the Test Results, and finally, the test results are parsed and stored by iterating through the JSON structure.

The data are committed to the database only when all information all processed correctly without raising a `sqlite3.OperationalError` exception. If the exception is raised, the data are not committed to the database and a a JSON object and HTTP status code `Internal Status Error 500` is sent to the client (with Flask `Response` object constructed by function `jsonify`).

Handling Waived failed results in a database

Waived results are handled by functions `get_waived_results()`, `add_waived_result()` and `remove_waived_result()`.
These functions receive an identification of a specific Book, Test Pack and Test Result that is to be added or removed from the database. These functions utilize helper functions to retrieve a specific primary key for a Book and Test Pack `get_bookname_pk()` and `get_testpack_pk()`.

Handling Database exceptions

The controller implements handling of `sqlite3.OperationalError` and `sqlite3.DatabaseError` exception. These exceptions cause the functions to return a negative value, which the main function represents in a JSON format and HTTP Error Code to the client.

Current implementation does not contain an optimal exception handling, so the database debugging has to be done in server's Python console, where the function name that caught the exception is displayed.

8.6 GUI with Test Results

HTMLs presenting the test results are generated with Jinja templates and are constructed by calling a Python Flask function `return_template`. The following section will describe the Jinja templates in more detail.

There are two documents generated with templates:

- `results.html` serves as a responsive dynamic website with presentation of test results of a specific revision of a book tested with a specific Test Pack. A single `results.html` document is regarded as a "test resource".
- `list_of_results.html` retrieves the list of available "test resources".

8.7 *results.html* Test Results Jinja Template

This template is used to render a dynamic and responsive HTML file. The HTML file includes a mechanism for reporting of false positive results by implementing a JavaScript jQuery library. The example of a generated HTML from the `results.html` template is presented in Figure 8.3

The template is formatted with Bootstrap framework and a supplemental CSS (Cascading Style Sheets) utilizing its grid system with containers, rows and navigation pillars.

The Test Suite tabbing functionality is implemented by the `tabbable` class from jQuery JavaScript library. The jQuery library is also used to issue AJAX calls to a waiving mechanism.

Sections of the *results.html* template:

- A header, containing the information about the book and Test Pack, sourced from the Test URI
- A row with tabs, containing Test Suite selectors with status bullets, signalizing the state of the Test Suite (Passed = Green bullet, Failed = Red bullet, No tests = Blue bullet)
- A row with a selected Test Suite, containing graph statistics about the Test Cases and foldable tabs, each containing a Test Case with a status bullet. The tab contains test results with a possibility to waive failed results.

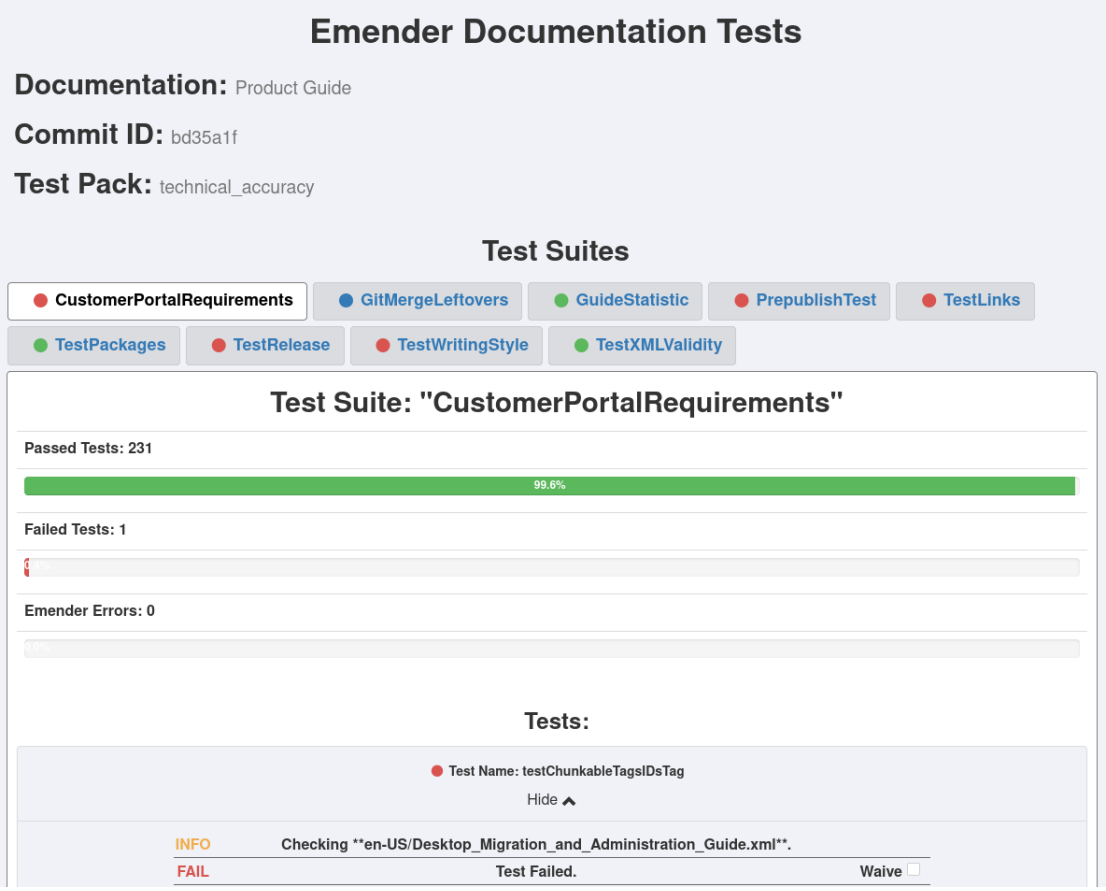

Fig. 8.3: emenderwebservice presentation of results

Sources of data for *results.html* template

Jinja template is populated by generating a HTML code by iterating through several Python dictionaries holding the data. These data are retrieved from the database and constructed when the user makes and API call to an endpoint `/api/results/<test_pack>`.

Sources of data:

- `json_dict`
  A dictionary that contains a JSON object converted into a dictionary. It is dynamically built by a script which retrieves the JSON from the database. (JSON is stored in the database as a string).

- `product`, `version`, `book`, `gitcommitid`, `testpack`
  Values parsed from the request's URI.

- `json_dict_graphs_suites`, `json_dict_graphs_tests`
  Dictionaries calculated dynamically and source data for test statistics.

- `waived_results`
  A list is generated by retrieving a list of waived failed results for a specific Book tested by a specific Test Pack.

Generation of test statistics

The statistics about tests are dynamically generated by `calculate_graphs` function that calculates the percentages of the graph rows and the states of the status bullets, stores it in dictionaries `json_dict_graphs_suites` and `json_dict_graphs_tests` for Test Suite and Test Case statistics. These dictionaries reflect the `json_dict` structure to simplify the identification of the graphs.

The graphs for individual Test Cases were not implemented, the dictionary holds bullet statuses displayed next to an individual Test Case.

8.8 Identification and Waiving of Failed Results

While the HTML with the results is being generated, the logic in template recognizes a failed test result and generates a HTML input field with a checkbox, with a unique identification (HTML `input` tag attribute `id`) next to the result. The template sets or unsets the checkbox depending on the results registration in the waiving mechanism by comparing the generated identification of the checkbox with the list in `waived_results`. Previous experiments included generating a form with *WTForms*[5] plugin.

When the user changes the value of the checkbox, a JavaScript included in the template sends an AJAX request to the REST endpoint `/api/WaivedResults/<test_pack>` with the unique identification of checkbox and its state. AJAX is only implemented for the waiving mechanism and user has to send another request to the API to receive an updated GUI (reload the page).

Asynchronous JavaScript and XML (AJAX)

"*AJAX enables web applications to send and retrieve data from a server asynchronously (in the background) without interfering with the display and behavior of the existing page.*"[22]

[5] `https://wtforms.readthedocs.io/en/2.3.x/`

9 Implementation Testing and Deployment

The testing of *emenderwebservice* application was done by running the application in a Flask development server and using Postman API testing platform[1], illustrated in Figure 9.1, to generate custom HTTP requests and analyze the responses.

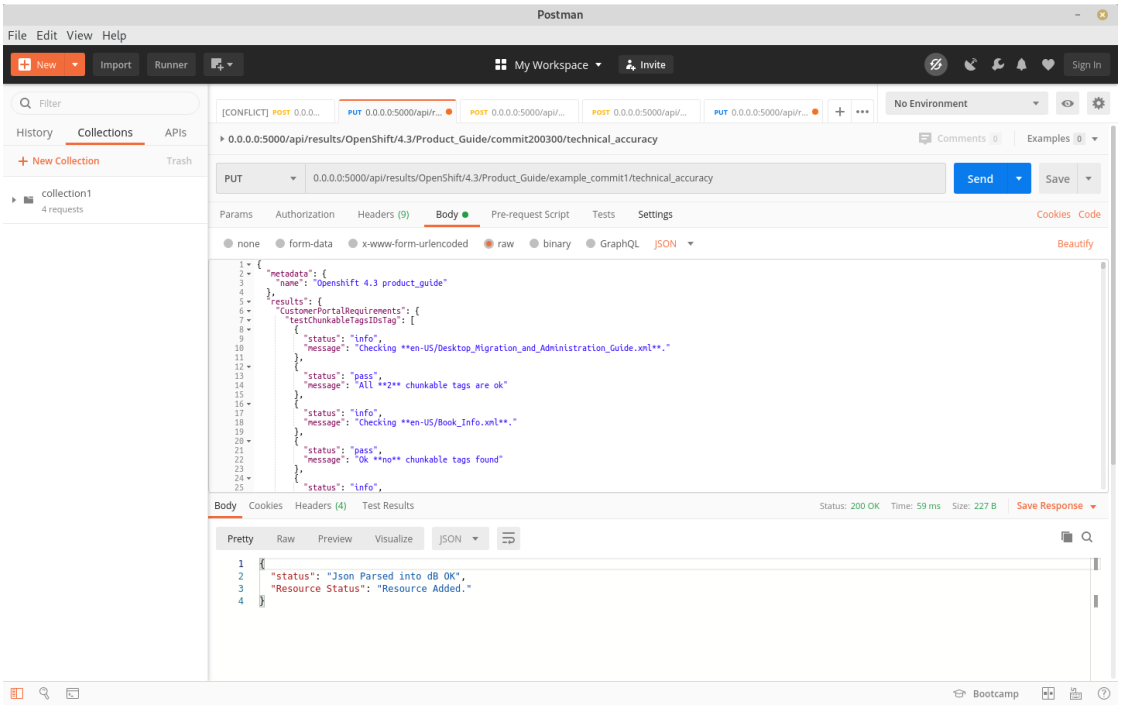

Fig. 9.1: Postman API testing platform

Testing JSON Schema

To test the JSON schema, various minimal changes were made to a valid JSON that made the JSON to be rejected with an error message in JSON format, which was then analyzed.

Testing database

Database was tested by sending several valid JSON files to the application and browsing the database with *DB Browser for SQLite* and visually, through the generated GUI with tes

[1] `https://www.postman.com/`

Testing the application with cURL

Application can be tested with cURL by sending a JSON file and retrieving the results, see Listing 9.1. A valid and invalid JSON file is included in the attached archive, folder `project_root/emenderwebservice/jsons/`.

Listing 9.1: Testing emenderwebservice with cURL

```
#!/usr/bin/env sh

# send JSON to emenderwebservice
curl -X PUT \
http://<server_ip>/api/results/<product_name>\
/<product_version>/<book_name>/<commit_ID>/<test_pack_name>\
-H "Content-Type: application/json"\
-d @<relative_path_to_json>

# retrieve junit xml
curl -X GET \
http://<server_ip>/api/junit/<product_name>\
/<product_version>/<book_name>/<commit_ID>\
/<test_pack_name> > results.junit

# see GUI with test results in a browser
<browser> \
http://<server_ip>/api/results/<product_name>\
/<product_version>/<book_name>/<commit_ID>/<test_pack_name>
```

9.1 Jenkins Configuration

The following section presents the example configuration of the Jenkins project. The example assumes that the Git repositories are set to correctly and contain a specific documentation. Listing 9.2 shows a configuration for *RHEL 4.1 Product Guide* tested with Test Pack named `technical_accuracy`. The Jenkins job first generates the test results files with a pre-configured Emender tests and sends a `results.json` file to the emenderwebservice REST API. The script then downloads and re-writes the `results.junit` file.

Because of the nature of the jobs, to incorporate the waived results into the Jenkins project (and process the JUnit XML file) a new build has to be triggered. (The author did not find a suitable way that would enable rewriting of existing build artifacts or re-running the same build).

Listing 9.2: Example Jenkins configuration

```
#!/usr/bin/env sh

./run_emend.sh

curl -X PUT \
http://<server_ip>/api/results/RHEL/4.1/Product_Guide/\
$GIT_COMMIT/technical_accuracy \
-H "Content-Type: application/json" -d @results.json

curl -X GET \
http://<server_ip>/api/junit/RHEL/4.1/Product_Guide/\
$GIT_COMMIT/technical_accuracy > results.junit
# $GIT_COMMIT contains a Git commit that triggered the build.
```

Implementation of project into a production server

Flask provides a development server, but using it for production is not recommended. A production-grade WSGI server is recommended, for example *Gunicorn*[2] Listing 9.3 shows an example how to run the application with the server.

Listing 9.3: Running emenderwebservice with Gunicorn server

```
#gunicorn <name_of_python_script>:<Flask_app_name>
gunicorn main:app
```

[2] https://gunicorn.org/

10 Suggestions for Further Project Development

Automated Jenkins job generation

The module emenderwebservice can be a starting point for a more complex web application. As stated in Section 4.3, the integration in Jenkins CI/CD requires user to understand the platform and its configuration, The adding of new documentation is not straightforward for the inexperienced users, even though the configuration would be very similar if not the same with the existing documentation. To solve this, a Jenkins REST API capabilities can be implemented into emenderwebservice to create new Jenkins jobs interactively.

Improved documentation for the project

The Emender project on GitHub is divided into several repositories. The repositories contain read-me files that document the implementation, but having a concise documentation, hosted on, for example GitHub Pages[1] might bring more popularity to the project. This thesis might serve as a suitable source of documentation.

Tutorials how to develop Lua tests

The tests developed for the framework cover a significant number of use cases. The documentation across different documentation teams might be structured differently and require test modifications. Currently, even though the source code for the test is well documented, the further development of tests is for the inexperienced user complicated.

Supporting pull request testing in Jenkins CI/CD

The current implementation is designed to test the main branches of the documentation. The process of submitting new documentation uses pull request method. Configuring Jenkins CI/CD to test pull requests would bring less incomplete contributions. The downside for the Jenkins CI/CD host would be a significant increase of builds that will require more server resources.

[1] `https://pages.github.com/`

11 Conclusion

The goal of the thesis was to implement an integrated framework for testing of technical documentation written in AsciiDoc markup language.

First sections discussed the aspects of writing technical documentation, presented several semantic markup languages, principles of Continuous Integration and Delivery and its role in technical writing. An open-source documentation testing framework Emender and its capabilities were introduced along with false positives in testing of technical documentation and a mechanism for their handling.

The thesis further discussed current implementation of Emender in an enterprise environment and a proposal of its extension with a RESTful web application providing test aggregation and a mechanism to handle false positive results. The implementation of the web application was discussed with a brief description of used technologies. The thesis was concluded with a testing of an implementation and suggestions for a further project development.

The major disadvantage of Emender project is an incomplete documentation, despite being extensively developed and successfully implemented onto enterprise documentation. This thesis tried to partly cover these shortcomings by providing an overview of functions and a description of implementation of the framework onto a technical documentation written in AsciiDoc. The implementation of *emenderwebservice* partly mitigates the problems with the current Emender implementation in CI/CD and provide a concise graphical user interface with test results aggregation in SQLite3.

The utilization of Flask web development framework in the implementation proved that it is ideal for a development of Python-based dynamic websites and provides excellent capabilities for RESTful services. Its extensibility enables further possible improvements of the application, notably an interface utilizing Jenkins REST API for documentation test management.

Implementation of *emenderwebservice* has a potential to bring more flexibility to the automated documentation testing process and may encourage new contributions to the Emender project.

Bibliography

[1] DERESPINIS, Francis. *The IBM style guide: conventions for writers and editors.* 2011. Upper Saddle River, NJ: IBM Press/Pearson, c2012. ISBN 978-0-13-210130-1.

[2] *Semantic | Definition of Semantic by Merriam-Webster* [online]. [cit. 2020-05-31]. Available at: `https://www.merriam-webster.com/dictionary/semantic`

[3] *Markup Language | Definition of Markup Language by Merriam-Webster* [online]. [cit. 2020-05-31]. Available at: `https://www.merriam-webster.com/dictionary/markup%20language`

[4] VOMÁČKA, Pavel. *Automatizovaná kontrola technické dokumentace.* Brno, 2017 [cit. 2020-05-31]. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce RNDr. Adam Rambousek, Ph.D..

[5] *AsciiDoc Home Page* [online]. [cit. 2020-05-31]. Available at: `http://asciidoc.org/`

[6] *Continuous integration vs. continuous delivery vs. continuous deployment* [online]. [cit. 2020-05-31]. Available at: `https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd`

[7] *Jenkins User Documentation* [online]. [cit. 2020-05-31]. Available at: `https://jenkins.io/doc/`

[8] *Web Services Architecture* [online]. [cit. 2020-05-31]. Available at: `https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`

[9] *HTTP request methods - HTTP | MDN* [online]. [cit. 2020-05-31]. Available at: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`

[10] *Are REST and HTTP the same thing? - The RESTful cookbook* [online]. [cit. 2020-05-31]. Available at: `http://restcookbook.com/Miscellaneous/rest-and-http/`

[11] *What is REST – Learn to create timeless RESTful APIs* [online]. [cit. 2020-05-31]. Available at: `https://restfulapi.net/`

[12] *JSON vs XML – REST API Tutorial* [online]. [cit. 2020-05-31]. Available at: `https://restfulapi.net/json-vs-xml/`

[13] *Foreword;Flask 1.0.2 documentation* [online]. [cit. 2020-05-31]. Available at: http://flask.pocoo.org/docs/1.0/foreword

[14] *Flask - Reviews, Pros & Cons | Companies using Flask* [online]. [cit. 2020-05-31]. Available at: https://stackshare.io/flask

[15] *PEP 3333 – Python Web Server Gateway Interface v1.0.1 | Python.org* [online]. [cit. 2020-05-31]. Available at: https://www.python.org/dev/peps/pep-3333/.

[16] *What is WSGI? — WSGI.org* [online]. [cit. 2020-05-31]. Available at: https://wsgi.readthedocs.io/en/latest/what.html.

[17] *Werkzeug — Werkzeug Documentation* [online]. [cit. 2020-05-31]. Available at: https://werkzeug.palletsprojects.com/en/1.0.x/.

[18] GRINBERG, Miguel. *Flask web development: developing web applications with Python. Second edition.* Beijing: O'Reilly, 2018. ISBN 978-1491991732.

[19] MANOLESCU, Dragos, Markus VOELTER and James NOBLE. *Pattern languages of program design 5.* Upper Saddle River: Addison-Wesley, 2006. ISBN 03-213-2194-4.

[20] *Overview of ASP.NET Core MVC* [online]. [cit. 2020-05-31]. Available at: https://docs.microsoft.com/en-us/aspnet/core/mvc/.

[21] *SQL Constraints* [online]. [cit. 2020-05-31]. Available at: https://www.w3schools.com/sql/sql_constraints.asp

[22] GUNASUNDARAM, Rajesh. *Learning Angular for .NET Developers.* Birmingham: Packt Publishing, 2017. ISBN 978-1-78588-428-3.

List of abbreviations

API	Application Programming Interface
CD	Continuous Delivery/Deployment
CI	Continuous Integration
CLI	Command Line Interface
FTP	File Transfer Protocol
GUI	Graphical User Interface
IDE	Integrated Development Environment
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MVC	Model-View-Controller
ORM	Object Relational Model
RDBMS	Relational Database Management System
REST	Representational State Transfer
SQL	Structured Query Language
URI	Uniform Resource Identifier
XML	eXtensible Markup Language

List of appendices

A Database Relationship Diagram

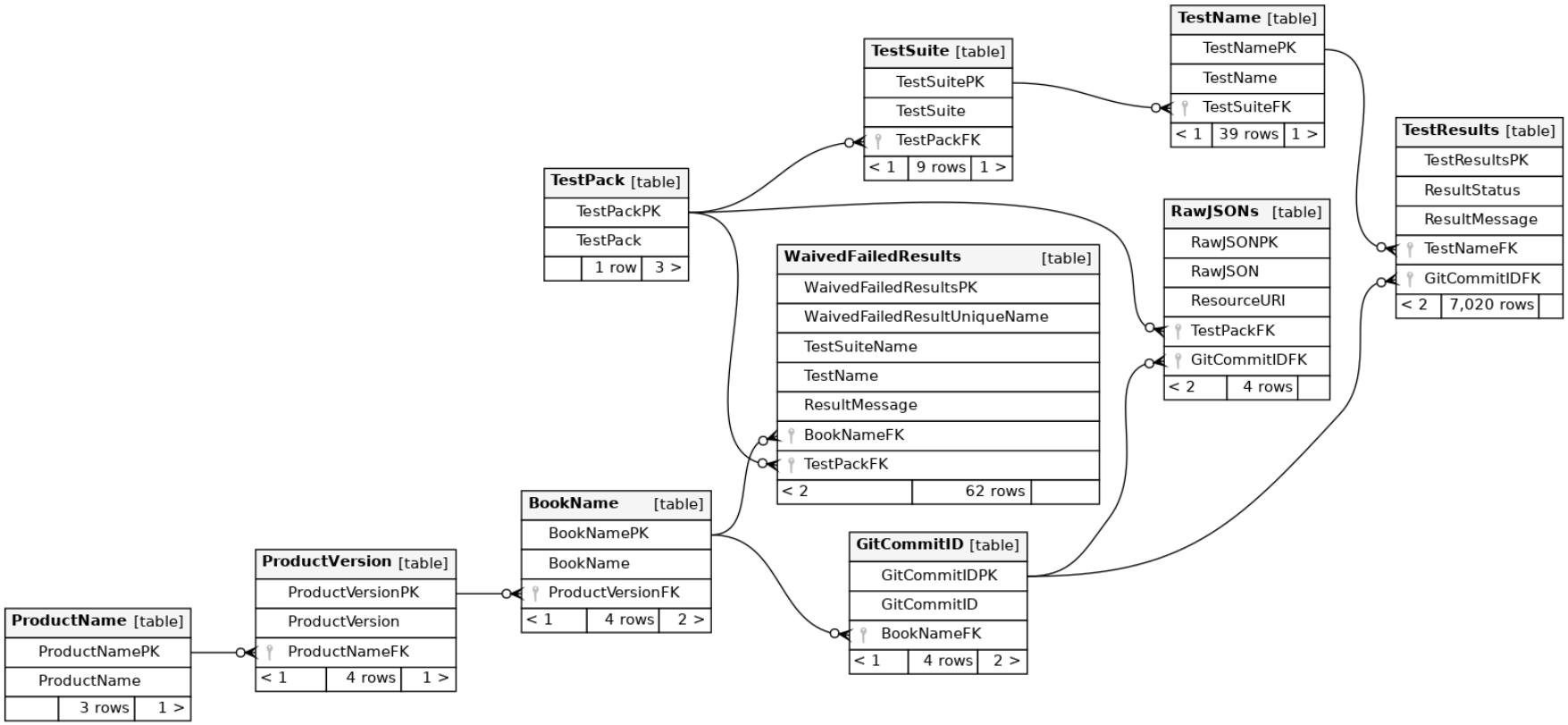

Fig. A.1: Database table relationships

B Attachment

B.1 Folder Structure

```
/
├── project_root/.....................Project root folder
│   ├── emenderwebservice/.....................Python module
│   │   ├── database/
│   │   │   ├── database.db.....................SQLite3 database
│   │   │   └── database-definition.sql
│   │   ├── jsons.....................Sample JSONs with test results
│   │   ├── static.....................Bootstrap, jQuery libraries and CSS
│   │   ├── templates.....................Jinja templates
│   │   ├── __init__.py.....................Module initialization
│   │   ├── calculate_graphs.py
│   │   ├── database_controller.py
│   │   ├── json_schema.py
│   │   ├── junit_rendering.py
│   │   └── routes.py.....................REST API definition
│   ├── flask_run.sh.....Bash script that runs a local server
│   ├── .flaskenv.....................Flask environment variables
│   ├── Pipfile.....................Dependency definition file
│   └── main.py
└── PeterMackoDiplomaThesis.pdf
```

B.2 Running the Development Server

1. Install `pipenv`.
2. Navigate to the `project_root` folder.
3. Run `$pipenv shell`. This installs the dependencies from the `Pipfile`.
4. Run `$./flask_run.sh`. This starts the development server and deploys the Flask application `emenderwebservice` on `0.0.0.0:5000`.
5. See Section 9 for application testing.

B.3 Project Structure

The project was implemented inside Python virtual environment, managed by `pipenv`. A folder structure recommended by Flask and Python was implemented, with the main Python package called `emenderwebservice`. The Flask application is defined in `main.py` which includes the Python package `emenderwebservice`.