BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

# TEST CASE MANAGEMENT WITH SUPPORT OF BDD
SPRÁVA TESTŮ S PODPOROU SCÉNÁŘŮ BDD

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. BARBORA BLOŽOŇOVÁ

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. ALEŠ SMRČKA, PhD.

BRNO 2019

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

# Zadání diplomové práce

21325

| | |
|---|---|
| Studentka: | **Blažoňová Barbora, Bc.** |
| Program: | Informační technologie Obor: Inteligentní systémy |
| Název: | **Správa testů s podporou scénářů BDD** |
| | **Test Case Management with Support of BDD** |
| Kategorie: | Analýza a testování softwaru |

Zadání:

1. Nastudujte projekty pro správu požadavků a testovacích scénářů (např. TestLink, TestCube). Nastudujte metodiku chováním řízeného vývoje programů (BDD, Behaviour Driven Development). Seznamte se s nástroji pro evidenci chyb a problémů.
2. Analyzujte požadavky pro správu požadavků, testovacích případů využívající scénáře BDD a reportování výsledků testování. Navrhněte informační systém pro správu požadavků a testů softwarových systémů. Systém by měl podporovat automatizované spouštění testů.
3. Implementujte navržený systém jako webovou aplikaci. Integrujte webovou aplikaci s jedním z běžně využívaných nástrojů pro evidenci chyb (např. Atlassian Jira).
4. Vytvořte automatizovanou testovací sadu pro všechny základní funkce webové aplikace.

Literatura:

- C. Solis and X. Wang. A Study of the Characteristics of Behaviour Driven Development. *2011. 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Oulu, 2011, pp. 383-387. doi: 10.1109/SEAA.2011.76
- Domovská stránka projektu TestLink: http://testlink.org/
- Domovská stránka projektu TestCube: https://www.utest.com/tools/testcube
- Domovská stránka projektu Jira: https://cs.atlassian.com/software/jira

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

| | |
|---|---|
| Vedoucí práce: | **Smrčka Aleš, Ing., Ph.D.** |
| Vedoucí ústavu: | Hanáček Petr, doc. Dr. Ing. |
| Datum zadání: | 1. listopadu 2018 |
| Datum odevzdání: | 22. května 2019 |
| Datum schválení: | 1. listopadu 2018 |

## Abstract

This thesis focuses on test management tools and automated testing. The project covers analysis of existing open source tools and proposes its own BDD orientated test management tool in the form of a web service. The project aims to specify and design this application based on the process of Behaviour driven development. The resulting application TestBuDDy allows for test library management. Changes on the test library are projected onto a remote repository of software under test (SUT) and triggers a test run (the test library is being run against SUT by the BDD framework). TestBuDDy is able to save the test run results, parse them into a report and generate and group found issues. The application also allows requirement management and user management. The application is integrated with the CI/CD tool Gitlab CI, the BDD framework JBehave and the issue tracker JIRA. The application is designed to help testers during their work and also to be expandable within the open source community.

## Abstrakt

Tato práce se zabývá prostředky pro správu požadavků a testovacích scénářů pro automatizované testování. Jejím cílem je na základě analýzy dostupných prostředků specifikovat a navrhnout webovou službu založenou na procesu Behaviour driven development, která pokryje jak správu požadavků testovaného softwaru, tak jeho automatizované testování. Výsledná aplikace TestBuDDy umožňuje správu testovací knihovny, kdy promítá provedené změny do vzdáleného repozitáře testovaného softwaru. Provedené změny spustí testy testovací knihovny na testovaný software (spravováno BDD frameworkem) a aplikace je schopna si interpretovat výsledky testů, uložit reporty a generovat a shlukovat nalezené chyby. Aplikace též umožňuje správu požadavků vůči testovací knihovně a správu uživatelů. Aplikace je integrována s CI/CD nástrojem Gitlab CI, BDD frameworkem JBehave a nástrojem pro správu chyb JIRA. Aplikace je navržena tak, aby usnadnila práci testerům, a s ohledem na budoucí expanzi v rámci open source komunity.

## Keywords

testing, test management, Behaviour driven development, automation, Continuous Integration, web service, information system, Gherkin syntax, requirement management

## Klíčová slova

testování, správa testů, Behaviour driven development, automatizace, Continuous Integration, webová služba, informační systém, Gherkin syntax, správa požadavků

## Reference

BLOŽOŇOVÁ, Barbora. *Test Case Management with Support of BDD.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, PhD.

# Test Case Management with Support of BDD

## Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Smrčka,PhD. with help of Daniel Šleis (Unicorn systems). All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . . . . . . . . . . . . . . . . . . . . .
Barbora Bložoňová
May 22, 2019

## Acknowledgements

I wish to express my gratitude to my supervisor and motivating advisor Ing. Smrčka,PhD. that always helped me with his insightful comments and feedback. I would like to also thank my Unicorn godfather Daniel Šleis for his support and knowledge from his test architect practice during the initial stages of this thesis. Last, but definitely not least, I would love to thank my family and friends for all the care, support and schnitzels that gave me the courage.

# Contents

# Chapter 1

# Introduction

With the gradual rise of technology usage in our daily life and growth in complexity of current IT systems, there is also a significant increase in demand for higher quality software. This need stems out the fact that it is almost impossible to guarantee 100% defect-free software and it is one of the key responsibilities of the whole development team to minimise such unexpected invalid behaviour. This is especially crucial in embedded systems that are becoming more and more common, not only in commercial industries such as power engineering or traffic control, but also in common modern households. While in the past, there was an unspoken rule that programmers did not need to be able to test software and testers were hired to only test the application with small technical background, the aforementioned current trends require industry response to rapidly deliver reliable, maintainable and secure software to the end customer as essential. Hence companies are heavily dependent on the overall quality of their software development process, used methodologies and the quality of software systems. In order to support new methodologies and approaches in software testing, the need for a new test management tool has emerged. The tool's primary functions are to manage software that is under test, support and maintain its testing cycle and support the cooperation of the testing team. The result application also has to connect testing requirements with the test results. The application is based on the *Behaviour Driven Development* (BDD) process. Another reason to implement this software development process is to bridge the gap between business and technical parts of the software development process [20]. BDD helps to resolve the communication issues between these two sides, and consequently saves time and effort. At present, there are many commercially available apps on the market such as *Practitest*[1], *qTest*[2], etc., but the targeted audience of this work is the *open source community* in which only a few test management tools are active and supported by its community to this day [1]. The project consists of an introduction, 7 main chapters and a conclusion. Chapter 2 discusses the required theory for the project which includes software testing, agile methodologies and finally puts these topics into context of automation. Chapter 3 analyses the existing test management tools and applications and gradually builds up the result requirements. Chapter 4 is analysing and comparing various programming languages and technologies that are used during implementation of the project. The technologies are examined with respect to the application architecture. Chapter 5, the application design, describes the application architecture, class diagram, scalability and data storage. This chapter also outlines possible implementation problems that are further

[1] Available at: `https://www.practitest.com/`.

[2] Available at: `https://www.qasymphony.com/software-testing-tools/qtest-manager/test-case-management/`.

addressed. Chapter 6 examines the implementation details concerning the application in general (endpoints, file structure, etc.) and describes implementation of application features, especially its test library. This chapter also connects the requirements, design and implementation, so that any potential users understand the project functions and flow. This connection for example, is apparent in figure 6.3 where user action and its sent data triggers multiple backend actions in relation to application design, BDD framework *JBehave* (integrated within the application) and requirements. The chapter is closed with details regarding application integrations *Continuous integration/Continuous delivery* tool *Gitlab CI* and an issue tracking tool *JIRA*. Finally chapter 7 describes application testing, its automated test suite and forms conclusions based on performed usability testing.

# Chapter 2

# State of the Art

This chapter introduces testing terminology and theory in order to put the context of software testing and agile methodologies into the project. This whole chapter puts emphasis on motivation to testing itself.

## 2.1 Software Testing

First, there are two terms worth mentioning to put testing into a software engineering perspective:

| | **Verification** | **Validation** |
|---|---|---|
| main focus | a process of examination and confirmation whether the tested software results conform to their set specified requirements | a process of examination and confirmation whether the results of tested software in a final development stage fulfil set specified requirements |
| development phase | all phases; the results are defined as preconditions to each development phase | final phases |
| aim | the product is being built in the right way | the right product was built |
| customer's presence | usually not needed | yes; as a response to customers' demands |

Table 2.1: Verification and Validation comparison

Verification and validation (V&V) recognise the following testing methods [9]:

1. Static analysis – oriented at the very form and structure of tested software without its execution.
2. Dynamic analysis – involves execution, or simulation of tested software component/feature/function in order to detect potential faults in software.
3. Formal analysis – uses rigorous mathematical models to quantify various aspects of tested software.

Software testing is a method of dynamic analysis of the tested software. According to book [7], software testing is a process to identify whether the tested software is correct, complete and of certain quality. In other words, it is an examination and analysis of the produced results against the expected results. These expected results are called **test requirements**. These requirements need to be set as part of the test design. Once the test or set of tests are designed, it is executed on tested software (this is called a **test run**), that subsequently produces a set of output values. The output values are then compared with test requirements and the result of the test is evaluated. The whole process of evaluating the test run outputs represents one way to analyse and measure the quality of the developed software in software testing.

### 2.1.1 Testing Vocabulary

Before fully looking into the next section, there are few terms from software testing terminology ([19]) to be explained:

- **Failure** (or a defect, an error, a *bug*) of software or computer program causes the program or software to produce unexpected incorrect results or behave in an unexpected incorrect way. Software faults are present since their implementation, but visibly manifest later when the software is actually executed. Book [7] recognises the difference between an error and a failure. Unlike an error that is viewed as an internal manifestation of a system fault, the failure is viewed as an external, incorrect manifestation of a fault with respect to the requirements in the specification. Defects are measured by **priority** which represents how big of a consequence there is on the developed software.

- **Test plan** ([7]) is a deliverable document for the customer that defines test objectives and sums up all the activities related to testing of the project. Test plan defines test strategy to be carried out, testing scope, schedule, roles and responsibilities, possible risks and their priority and finally input and output criteria. Test plan can also be split according to each stage of a development cycle.

- **Test strategy** describes testing approach used for selected tested software.

- **Test suite** is a set of test cases based on a common logical unit, or a feature on which a test run will be executed.

- **Test case** is a test defined by a set of inputs (conditions, variables) and expected outputs.

- **Test scenario** is a set of test steps. Test scenario is part of a test case.

- **Test step** is an action to be executed with set inputs (preconditions) and outputs (results).

At first, a test suite is designed, test data is prepared, then the tested software is executed with this test data and finally, the test run results are compared to the expected results from the test suite.

### 2.1.2 Software Development Life Cycle and a Cost of a Defect

This section helps to further demonstrate the importance of testing in different stages of a product development cycle. The whole process between the initial thought of a product until its final delivery to the customer is known under the abbreviation SDLC, or *Software development life cycle*. SDLC can be divided into several stages as the process involves many tasks. SDLC's main purpose is to improve quality of developed software and overall development process [18]. At present there are multiple known models to deliver software that mainly stand apart by different sequencing of their phases. The phases can run sequentially or simultaneously. From the testing point of view, the **V-model** was introduced and unlike fully sequential *waterfall* model [17], the V-model plans the testing phase in parallel to each development phase of its cycle. Although the V-model is viewed as an extension of its predecessor, it clearly distinguishes the connection of each phase of development with its respective level of testing. The model activities are split between verification and validation. Another cause for introducing V-model is the fact that as its stages are progressing during the software's life cycle, the relative cost of fixing found *defects* has a logarithmic dependency towards the time [14]. This fact was already known and measured in the past ([12]) and lead to the popularity of V-model over waterfall.

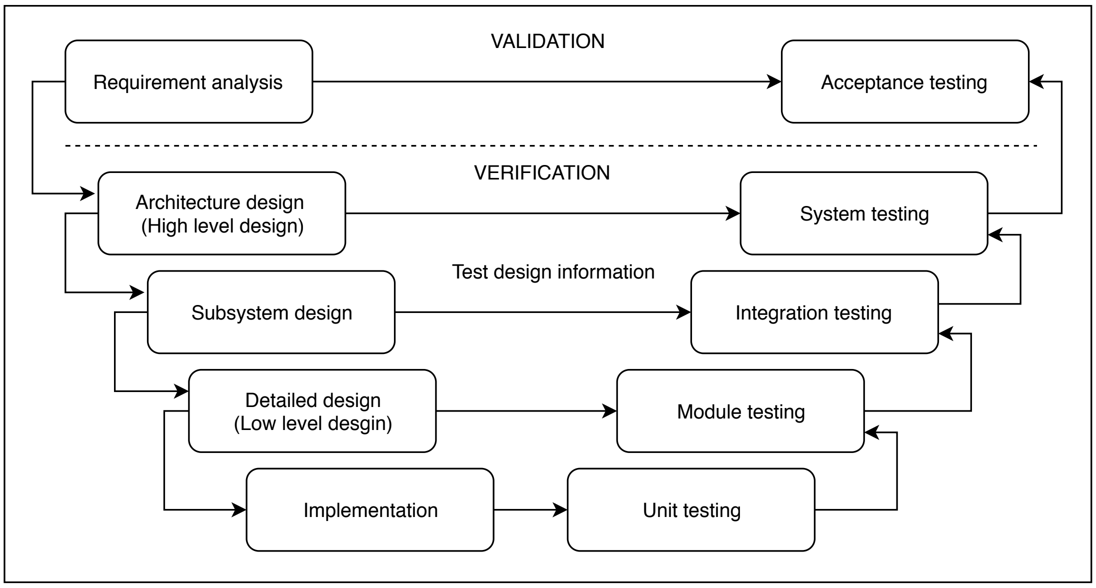


Figure 2.1: V-model in the context of V&V ([7]). The *test design information* needed for each testing phase refers to a test plan with test cases for each phase, etc.

Based on V-model, there are 4 main levels of testing:

1. **Unit testing** is the lowest level of testing that examines and measures the most basic functions of software components with respect to their *Low level design* (LLD). Unit tests are usually executed by developers. The aim is to discover defects due to the lowest cost of bug fixing at this level.

2. **Integration testing** that assesses subsystem modules and their communication with respect to their *Subsystem design*. The common sought-after defects are errors in interface communication.

3. **System testing** that assesses the system as a whole with respect to its *High level design (HLD)*. The aim is to find defects in the architecture design and requirements.
4. **Acceptance testing** is a phase to validate and assess whether the completed software fullfills its requirements from *Requirement design*. In case of client presence during this phase, the term *user acceptance testing (UAT)* is used.

## 2.2 Agile Methodologies

Although V-model brings parallel planning, the reality of this high-discipline model is still very sequential and rigid for software development. V-model is not suitable for projects of such dynamic nature where there is a high risk of changing the requirements and the whole project has to start its cycle anew. The changes are subsequently very time and cost demanding which eventually lead to ”the software development crisis” during the 1990s. The crisis gave rise to seek more ways to deliver software rapidly, so the *agile methodologies* eventually emerged and had been commercially popularised.

### Behaviour Driven Development

The main concept of **Behaviour driven development (BDD)** as a software development process is to bridge the gap between business and technical point of view on tested software [20]. BDD essentially tries to take the best practices from its ancestor *Test driven development* (TDD). TDD's main concept is to write a test before the source code, so the test initially fails, until the developer provides the required implementation. On the other hand, BDD puts emphasis on behaviour of the source code features by defining **user stories** and using ubiquitous language [16]. *Domain specific language (DSL)* uses constructs from the natural language so that the customer and project manager can understand the name and content of each test. User story is characterised as:

Demonstrative example of a story

```
As a role
I want a feature
So that benefit
```

By ”benefit” it is meant a benefit for the customer. Each test scenario is first written and narrated as a story. This approach is heavily dependent on correct requirements specification, therefore BDD also introduces a **gherkin language** and tools. Gherkin syntax has ”Given-When-Then” pattern - a demonstrative example shows these keywords in blue color:

Example of gherkin syntax

```
Scenario/Feature: Descriptive clear name
Given Precondition/situation I'm at
When I do action
And I do another action
Then I expect results of these actions
```

One line represents the **BDD declaration** of a step. Gherkin also supports parameters, batch runs and other features. The story is then mapped to executable code of an automated test, in other words a **BDD definition** is added to its BDD declaration. The process of mapping is usually done with the use of a BDD support tool, or **BDD framework** [16]. Finally, a test run is executed and the final result exported into a test report.

### 2.2.1 Why to Test

The term ”quality” is often mentioned in the context of software. It is crucial to understand that there is not one correct way to measure quality [7]. While the *test team* can only look for a presence of a defect, there is no way to actually prove an absence of a defect. A **test engineer** is a part of a development team and his fundamental job is:

1. to discover and report errors in software at the earliest stage possible to prevent late costly changes in the product (this is done by an early installation and execution of a test phase during *Software Development Life cycle*)
2. to communicate with developers that the tested functionality is defect-free
3. to communicate with management in terms of potential risks for the product and internal processes
4. to communicate with the customer that the product behaves in a correct defined way and adheres to its requirements

Together the test manager and a test engineer form a test team. These activities also show the reasoning to have a tester role during the early stages of the product development cycle [1]. The results of all of these activities lead to main testing motivation: to deliver software of as high quality as possible to the end customer.

## 2.3 Automatio of Testing

Another important aspect of high quality software is its changeability. When the code changes during SDLC, for instance by adding a new feature, the whole system has to be thoroughly and repetitively tested to ensure that the new code has not brought any new defects or did not negatively influence the rest of the system, this is called *regression testing.* Although the initial budget of **test automation** is higher than manual regression testing, an automated approach is more efficient and effective in the long run (as seen in [9, Chapter 4]). Alongside agile practices, the *development operations*, or ”**DevOps**” are being used in daily practice due to 2 main needs: not only to develop, but also to deliver and deploy the software as quickly as possible and to allow smaller but more frequent changes in software. *Continuous Integration (CI)*, *Continuous delivery (CDE)* and *Continuous deployment (CD)* define a set of tools and practices to mitigate discontinuities between development, delivery and deployment [13]. Overall the key principles of DevOps as a set of practices and tools are defined by the following key practices:

- having source code in a remote repository with use of a **version control** to **integrate** developer’s new or changed code with remote source code of the produced software; a part of the CI.
- faster, more frequent releases in forms of **builds** after smaller changes; this practice belongs to CDE[2].
- an automated test suite[3]; also a part of the CDE approach.

[1]This fact is known as ”early testing”.
[2]While the CDE approach allows us to build and deploy software through manual release, the CD’s approach offers a continuous automated build process and deployment.
[3]The automated test suite is executed in an amount of time that the team productivity doesn’t drop [13].

Chapter 3

# Analysis of Existing Applications and Requirements

This chapter is divided into several sections: part one introduces, classifies and analyses *test management tools*. Part two examines and compares available tools. Part three will build upon this analysis and contains the result requirements for the project application and finally section 3.4 discusses tool integration with other technologies.

## 3.1 Test Management Tool

According to [19, chapter 7], a *test management tool* ensures product specification, execution, prioritisation, categorisation and overall maintenance of test cases. Furthermore, requirements can be linked to each test case. The main areas of focus for advanced test management tools are: **requirements management**, **test specification**, **test execution**, **test planning**, **incident management** (to manage found failures) and **configuration management** (to keep track of different versions and builds of the developed software). According to **software licensing**, there are 2 main groups:

- **proprietary/commercial**, where the customer has to pay for the product. These tools are usually paid as a **service** by means of subscription. The examples are many, as of 2019, the most used are: *Practitest*, *Microfocus ALM* (known in the past as *HP Quality center*)[1] and the hybrid example, *HipTest*[2] where the product is free for open source projects, otherwise commercial.

- **open-source**, where the product is free in its full available version. As of 2019 [1], the examples are *Testlink*[3] (the most used) and unfortunately closed down project *Tarantula*[4].

The goal of this project is to design an open-source basis of a tool comparable to its commercial counterparts. The tools can be also classified by their **dependency on installation**:

- **web-based application**, where the app does not need an installation and is accessible through a web browser. These applications can be further divided between

[1]Available at: `https://www.microfocus.com/en-us/products/application-lifecycle-management/overview`.

[2]Available at: `https://hiptest.com/`.

[3]Available at: `http://www.testlink.org/`.

[4]Available at: `http://www.testiatarantula.com/`.

cloud-based (application as *service* is operated remotely and a client can only access the application) and client-based (client operates the application on his side). Example of cloud-based applications are *HipTest* and *Practitest*, an example of client-based application is *Testlink*.

- **non web-based or desktop application**, where the application needs to be installed on a local machine; example is *Microfocus ALM*.
- a hybrid, or the applications that are offered depending on a client's offer. An example is *qTest* that offers both desktop and web-based solutions.

From the approach towards the **incident management** there are apps offering integration to existing *issue trackers*, or implementing their own issue tracking. These issue trackers are further compared in section 3.4.2.

## 3.2 Comparison of Existing Test Management Tools

To better understand and set requirements for the project, the 3 competitors are compared in table 3.1. *Testlink* was selected as the leading open source tool, *Practitest* as one of the leading commercial software thanks to its number of active users and wide offer of features and finally *HipTest* as a hybrid tool that is primarily orientated on BDD. The examined categories are based on analysed basic functions of test management tools from the previous section and serve as a base to form a full list of application requirements in the next section.

| | **Testlink** | **Practitest** | **Hiptest** |
|---|---|---|---|
| **license** | open source | proprietary | free for open source projects; otherwise proprietary |
| **project management** | management of multiple projects | management of multiple projects | management of multiple projects |
| **requirements management** | requirements and event logs; requirement mapping to test cases | requirements and event logs; requirements mapping to test cases | native BDD support; requirements in documentation |
| **test planning** | test plan management and versioning | test plan management and versioning | no |
| **test specification** | test case creation; assigning keywords (as library); test suite management, maintenance; test runs | advanced test management and filtering | scenario editor; reusable steps and step auto-completion and re-factoring |
| **test execution** | manual and automated test execution | test runs and cycles; versioning; manual and automated test executions | data sets creation; test creation and generation |

| | **Testlink** | **Practitest** | **Hiptest** |
|---|---|---|---|
| **test reporting** | report generation; metric support; export | advanced reports and dashboards; metrics support; export | advanced reports and dashboards; advanced metrics for each test; export |
| **configuration management** | build management | CI tools integrations | CI tools integrations |
| **incident management** | defect recording; integrated with common issue trackers (JIRA, Bugzilla,...) | fully integrated with various issue trackers; anti-bug duplicates and calls to test | integrated with JIRA as a plugin |
| **version control** | yes | yes, including Git, SVN, ... | yes; using local service HipTest publisher |
| **automation integration** | integrated, but limited | PractiTest's API; fully integrated with automation tools as Selenium | fully integrated with 20+ automation frameworks such as Cucumber, Java/JUnit, Selenium |
| **user and role management** | yes; integrated various roles and test assignments | yes; integrated various roles and test assignments | limited to administrator and users; integrated test assignment |
| **running environment** | web-based; client-based | web-based; cloud-based | web-based; cloud-based |
| **price** | free | $35-45/month | $24/month (with BDD support) |

Table 3.1: Comparison of selected test management tools - functionality requirements and basic information.

## 3.3 Application Requirements

[17, Chapter 4] splits *software requirements* into the following categories:

1. **user requirements (UR)** that are usually written in natural language for a non-technical audience.

   1.1. **functional** that describe the functionality/behaviour of the system.

   1.2. **non-functional** that represent constraints on product organisational and external properties such as performance.

2. **system requirements (SR)** that describe system external behaviour in a technical manner.

First, the user requirements are presented in detail on which the system requirements are then built upon. The final list of requirements is therefore in section 3.3.3.

### 3.3.1 User Requirements - Functional Requirements (FR)

1. **User and role administration:**
    1.1. assign test team users to a project,
    1.2. create, edit, manage user accounts,
    1.3. create, edit, manage team roles,
    1.4. connect role(s) to a user account.

2. **Project management**:
    2.1. create, edit and remove projects,
    2.2. edit project information e.g. description, name, schedule, scope, etc.,
    2.3. filter and list projects.

3. **Test planning**:
    3.1. create, edit and remove test plans and their information e.g. test plan version, description, etc.,
    3.2. assign test cases to a test plan,
    3.3. assign test plan to a project.

4. **Test design** (test specification)[5]:
    4.1. create, edit and remove test modules[6] in a test plan,
    4.2. create, edit and remove test cases,
    4.3. assign test cases to a test module,
    4.4. add and manage test case information,
    4.5. automatically edit test step definitions[7] based on their usage in test module within test plan for a project (test step definition are to be stored as a library for a project),
    4.6. create, edit and remove a test step[8] within the test case. Each test step is based on a test step definition and added input parameters ( = **BDD step declaration**[9] that is connected to its **BDD definition** within BDD framework),
    4.7. filter and list test step definitions, test modules and their test cases.

5. **Test execution** (test runs) management, test cases execution):
    5.1. execute test run and save its results (possible results are: *Success*, *Failed*, *Pending*, etc.)
    5.2. **automated testing**:

[5]All of the following actions are to be in sync with remote repository.
[6]A *test module* represents test cases folder structure in a test plan.
[7]A *test step definition* is the base of a test step.
[8]A *test step* is a template with concrete parameters.
[9]The expected results of a test step are saved as another BDD declaration in gherkin syntax: `Then [BDD declaration of the result]`.

5.2.1. add **BDD step declarations**[10] to remote repository (via used test steps in test case),
5.2.2. generate BDD declarations for the scenario test steps with failing, **BDD definitions**, save the changes on remote repository, thus immediately issue a test run (that fails),
5.2.3. issue a new test run (trigger a rerun),
5.2.4. view, filter and list results of all test runs.

6. **Incident management**:
   - 6.1. automatically create bug out of failing test run (based on failing BDD step definition),
   - 6.2. connect to an issue tracker to monitor bug status,
   - 6.3. assign a bug to a failed test step definition (used in test cases),
   - 6.4. list and filter bugs for project.

7. **Test reporting**:
   - 7.1. view test run reports from **triggered automation**[11],
   - 7.2. save test cases results and overall result of finished test run into a report,
   - 7.3. filter and list reports.

8. **Requirement management** - to ascertain test coverage as a metric:
   - 8.1. create, edit and remove requirements,
   - 8.2. link requirement to a test case,
   - 8.3. tag requirement and manage tags for project,
   - 8.4. filter and list requirements.

### 3.3.2 User Requirements - Non-functional Requirements (NFR)

1. The application should be **multiplatform**.
2. **Performance**: The application is designed for testing teams ranging in size from smaller teams to tens of members.
3. **Executability**: The application should be implemented as a **web service**.
4. **Source code maintainability**: The source code is to be written in a structured, documented way.
5. **Extensibility**: The application's architecture should be designed for future extensions and changes respecting the integrations (CI/CD tool, Issue Tracker, etc.).
6. **Pricing/licensing**: The application's target audience is an open source community.

[10] Mapped source code to its respective BDD declarations.
[11] Automation is triggered by committed changes on a remote repository of the developed software by CI/CD tool, this is discussed in chapter 4.

### 3.3.3 System Requirements

This section connects user requirements to the final table of application requirements with the use of **application Use case diagram**. The system requirements are modelled and simplified into UML *Use case* ([17])[12] and a summary list of system requirements. Use case connects system requirements and their respective detailed user requirements from the previous section. The use case in figures 3.1, 3.2 and 3.3, displays 3 main recognised groups, each group represents 1 or more functionality groups of the application (from the requirements in section 3.3) and is visually framed with a coloured rectangle. As seen on figure 3.1, although the application recognises user roles (for example an *administrator*, a *test manager*, and a *tester*) the Use case was mainly created for a general user.

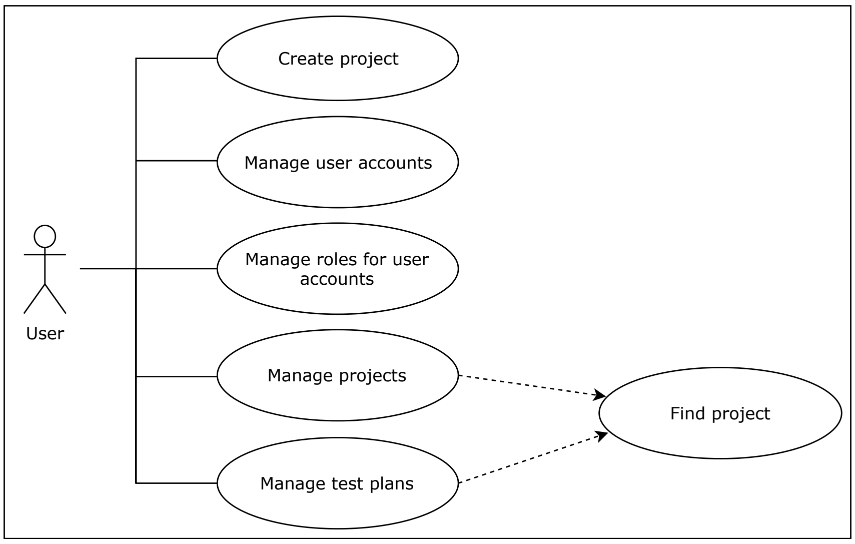


Figure 3.1: Section 1 of the application Use case: *User and role administration, Project management and Test planning* groups.

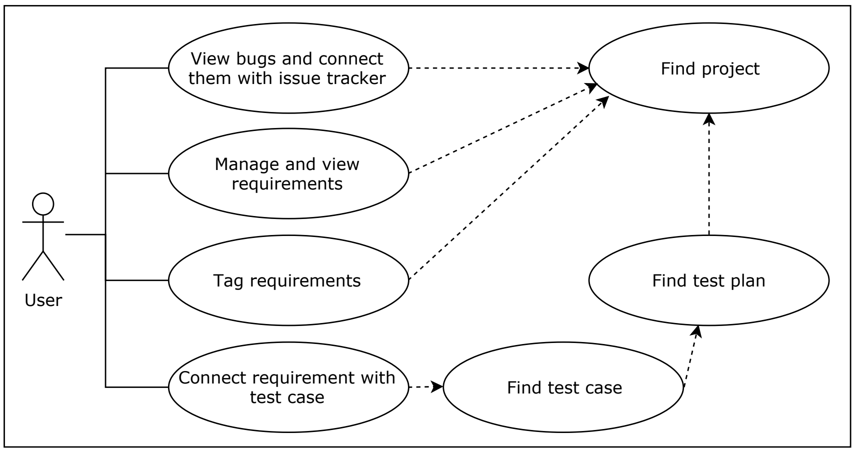

View bugs and connect them with issue tracker
Find project
Manage and view requirements
User
Tag requirements
Find test plan
Connect requirement with test case
Find test case

Figure 3.2: Section 2 of the application Use case: *Requirement management and reporting* groups[13].

[12] Dashed arrow lines represent `include` relation.
[13] *Incident management* group from the user requirements is included and simplified within the actions of this group.

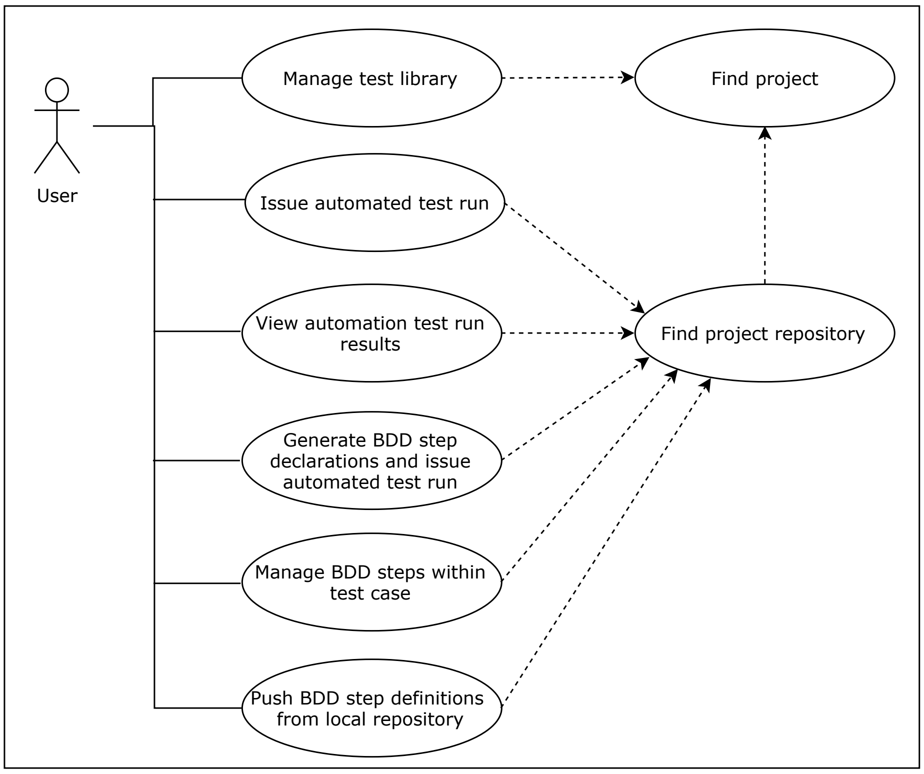

User
Manage test library
Find project
Issue automated test run
View automation test run results
Find project repository
Generate BDD step declarations and issue automated test run
Manage BDD steps within test case
Push BDD step definitions from local repository

Figure 3.3: Section 3 of the application Use case: *Test specification and Test execution*

The resulting list of final application requirements is displayed in table[14] 3.2.

| **Code** | **Requirement** | **Group** | **Dependency** |
|---|---|---|---|
| FR-01 | Application supports user and role management. | User and role administration | |
| FR-02 | Users can manage multiple projects. | Project management | FR-01 |
| FR-03 | Users can manage and view test plans. | Test planning | FR-02 |
| FR-04 | User can connect a test plan to a project for future test run. | Test planning | FR-03 |
| FR-05 | Users can add test modules to the test plan and test cases to the test module. | Test planning | FR-03<br>FR-08 |
| FR-06 | Users can manage and view the test library (test modules, test cases, test steps, test step definitions). | Test specification (design) | FR-05 |
| FR-07 | Users can add test steps to test case (that triggers FR-13). | Test execution | FR-05<br>FR-08 |
| FR-08 | User connects project to remote repository (managed by CI/CD tool) and his future actions in thesis application are visible in connected remote repository. | Test execution | FR-02<br>FR-04 |

[14]The column *Group* classifies each requirement into context of groups from previous sections. *User* contains all roles such as tester, test architect or administrator.

| **Code** | **Requirement** | **Group** | **Dependency** |
|---|---|---|---|
| FR-09 | Users can view generated bugs from failed test step definitions based on affected test cases. | Incident management | FR-05<br>FR-08 |
| FR-10 | Users can link an issue tracker to a bug. | Incident management | FR-09 |
| FR-11 | Users can view results of a test run. | Test reporting | FR-07<br>FR-08<br>FR-10<br>FR-13 |
| FR-12 | Users can issue an automated test run = trigger a rerun. | Test execution | FR-08 |
| FR-13 | Users can generate BDD step declarations based on the test library. | Test execution | FR-06<br>FR-07<br>FR-08 |
| FR-14 | Application commits generated BDD step declarations as changes into the project repository and triggers an automated test run. | Test execution | FR-08<br>FR-13 |
| FR-15 | Users can view results of triggered automation (a change was commited on the project repository). | Test reporting | FR-08<br>(FR-12) |
| FR-16 | Users can manage requirements and their tags. | | FR-05 |
| FR-17 | Users can link requirements to test cases and tag them. | | FR-16 |
| NFR-01 | The application's architecture design should be scalable in case of future extensions. | non-functional | |
| NFR-02 | The application should be integrated with a chosen CI/CD tool, a BDD framework and an issue tracker. | non-functional | NFR-01 |
| NFR-03 | The application should be multiplatform as a web service. | non-functional | |

Table 3.2: The final table of requirements.

## 3.4 Application Integration

This section discusses the comparison of possible application integrations from diverse fields.

### 3.4.1 Comparison of Existing CI/CD Tool Providers

With reference to FR-08, FR-14 and this thesis orientation on automation, the CI/CD tools that allow automation and version control has to be decided. As of 2019, amongst top open source tools, there are 3 selected CI servers to be evaluated: *Gitlab*[15] with support of its CI *Gitlab CI*[16], *Jenkins*[17] and *Travis CI*[18]. Jenkins is one of the biggest open source CI tool with a large support of various plugins. Jenkins also comes with a support of open-source community and can be heavily customised to one's needs. Travis CI offers its product not only as a service, but also offers the possibility of its installation on client's premises. While Jenkins is fully open-source, Travis CI is free only for open-source projects. Although the thesis' resulting application is meant to be open-source, its project repositories don't have to necessary be the same. According to [8] report, Gitlab CI provides wide range of DevOps tools and, although relatively new, has the strongest and still rising market presence and was chosen as a leader in Continuous Integration. GitLab CI basic functionality is open-source with full support for either small or large scale products [8]. In the end, Gitlab CI was chosen, since *Gitlab CI* complies with the thesis' requirements and shows high potential in future.

### 3.4.2 Comparison of Existing Issue Trackers

Regarding the FR-10 requirement and overall thesis assignment, the integration with an issue tracker needs to be decided. The lead existing issue trackers ([2]) are *JIRA*[19] (for commercial use) and open source issue trackers *Bugzilla*[20] and *Redmine*[21]. Trackers were mainly compared by their popularity (by number of professional job offers in the IT field, number of website mentions and so on). My main requirement was to choose the most used tracker (as of 2019) that is also a web service ergo, I've chosen *JIRA* because this commercial issue tracker by far surpasses its open source competitors [5].

### 3.4.3 Comparison of BDD Frameworks

With respect to FR-12 and FR-14, I need to also choose a suitable BDD framework to be integrated into the application. Usual flow of BDD framework is: write a story; map it to the framework supported executable code (usually Java); configure which tests to run; run the tests and view the report. Chosen open source competitors are: *JBehave*[22], *Cucumber*[23] and *Concordion*[24], these competitors were compared based on different categories on table 3.3.

[15] Available at: `https://about.gitlab.com/`.
[16] Available at: `https://about.gitlab.com/product/continuous-integration/`.
[17] Available at: `https://jenkins.io/`.
[18] Available at: `https://travis-ci.com/`.
[19] Available at: `https://www.atlassian.com/software/jira`.
[20] Available at: `https://www.bugzilla.org/`.
[21] Available at: `https://www.redmine.org/projects/redmine`.
[22] Available at: `https://jbehave.org/`.
[23] Available at: `https://cucumber.io/`.
[24] Available at: `https://concordion.org/`.

| | **JBehave** | **Cucumber** | **Concordion** |
|---|---|---|---|
| **project language** | Java based projects | various languages | various languages |
| **automation** | desktop and web; support for JUnit tests | only web-based | desktop and web |
| **story format (stories, scenarios)** | text statements | requirement statements, test conditions | HTML specification |
| **Gherkin format** | yes and more | yes | yes |
| **mapping code** | Java | various | Java |
| **JUnit support** | yes | yes | yes |
| **documentation** | extensive; but rather hard to read | extensive; up to date | poor; but up to date |

Table 3.3: Comparison of BDD frameworks.

Since *Unicorn*[25] uses *Java*[26] as their main programming language for power engineering projects, my choice was based mainly on the features of each framework. In the end *JBehave* was chosen, because: it supports not only a pure BDD approach in the gherkin syntax[27], but also possesses many more features. Whilst *JBehave* is the oldest of these 3 frameworks, it is still very popular.

[25]Company's web pages: `https://unicorn.com/cz/`.

[26]Java SDK is available at: `https://jdk.java.net/`.

[27]Article [16] compares different BDD frameworks based on their support of the BDD characteristics. *JBehave* supports majority of these recognised characteristics.

# Chapter 4

# Analysis of Used Languages and Technologies

The goal of this chapter is to analyse and decide programming languages and technologies that are to be used in thesis implementation. The chapter is split into 2 parts: 4.1 that discusses web applications and their composition, 4.2 that covers technologies for backend development of the application that serves as the base of the resulting web service.

## 4.1 Web Applications

Based on NFR-03, the project will be implemented as a *web service*. As the *World wide web* evolved during the last 30 years, it allowed us to develop more complicated web applications [17, Chapter 1]. The current trends indicate the rising use of web development tools known as *web application frameworks*. A framework's main purpose is to provide a set of libraries and tools that help to build a modern web application that can be further reused.

### 4.1.1 Backend and Frontend

General application architecture consists of client side, also called a frontend, and server side, known as a backend. Client side views and interacts with the web page, whereas server side consists of a server reacting to user's requests, the application and its logic and finally data storage in the form of a database and connection to it. These functions are described in detail in section 5.1. Thesis application is to be implemented as a *service*, so it mainly focuses on the **backend** side.

### 4.1.2 Model - View - Controller (MVC)

MVC is a popular architectural pattern based on division of the application logic and interface. MVC splits the application into 3 sections:

- **model** is a central section that represents application logic and its data. Model receives updates from the controller.
- **view** is a rendered representation of the model.
- **controller** receives and processes user input and decides what is then passed onto the model or the view.

Figure 4.1 puts MVC pattern into a context of backend and frontend.

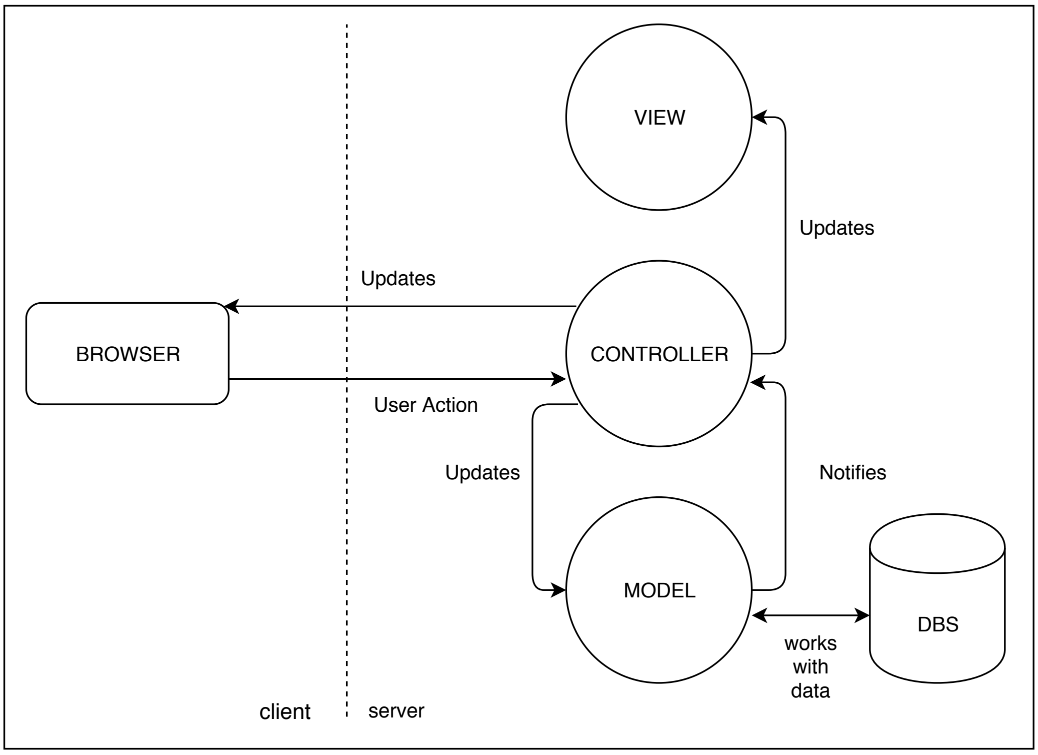


Figure 4.1: MVC versus client and server. ”User Action” is usually in form of *HTTP Request* from a client browser to the server, the server then sends back *HTTP Response*, this is more discussed in section 5.1.

### 4.1.3 ORM and Data Storing

The data is to be stored in relational database in the form of tables. Database is managed by *Database Management System* (DBMS). One of the leading DBMS is *MySQL* [3] that uses *Structured Query Language* (SQL). To connect a database to programming code, data needs to be *linked* to the language abstraction. *Object-relational mapping* (ORM) is a practice to map data from relational database to OOP (*Object oriented programming*) language.

## 4.2 Backend Framework

Before the actual choice of framework, a programming language for the backend of the application needs to be decided.

### 4.2.1 Chosen Programming Language

**PHP**[1] is an open source general purpose and server-side scripting language which is unfortunately not suitable for large scale applications. Key features of this robust, easy-to-learn language are: stability and support for various libraries and speed. PHP also offers full integration with popular databases like *MySQL*. Unfortunately, PHP used to be the most

[1]Available at: `https://secure.php.net/`.

popular over the internet, but from [6] it is apparent its popularity - although large - is steeply decreasing.
Another candidate is **Java** as this language is used by *JBehave* and has support for web applications. Java is a general purpose language that also supports a large variety of in-built frameworks like *Spring*, *J2E*, etc. Java source code is compiled into a byte code and then interpreted by *Java Virtual machinde*, or JVM. Java offers not only OOP approach, but also orientation on security (as opposed to PHP) and scalability suitable for large scale applications. Java overall (dis)advantages are:

+ large range of addable frameworks
+ performance; JVM
+ built-in memory management
+ large developing community

– verbose coding style
– hard to set up at the beginning
– dependency on IDE (*Integrated development environment*)

On the other hand the last candidate, **Python**[2], is the fastest growing in use as of 2017 [15]. Python is a high-level general purpose, interpreted[3] and object oriented programming language. Python's main advantages are:

+ easy-to-learn
+ organised and readable code
+ extensive well-documented library
+ easy package management
+ offers a variety of additional modules, plugins and libraries for web development

– in general as an interpreted language: slower than PHP

The key priorities for the final choice were the learning curve and the potential to be extended in future, the best possible choice for language was **Python**.

### 4.2.2 Chosen Python Framework

From a framework point of view, there are several frameworks to be taken into consideration: *Django*[4], *Pyramid*[5] and *Flask*[6]. Although Django offers out-of-the-box support of ORM and wide range of modules for web development, compared to Flask it is more demanding by means of memory and space usage. On the other side, there is a microframework, Flask, that does not need a bootstrapping tool and is more orientated on the programmer's own choice of storing data and which components to choose for your application. As of 2017 [10], Flask has gotten more popular than Django on *StackOverflow*[7]. For these reasons, I've chosen Flask as my backend framework. Here is a summary of its advantages:

[2]Available at: `https://www.python.org/`.
[3]Python does not need to compile before running.
[4]Available at: `https://www.djangoproject.com/`.
[5]Available at: `https://docs.pylonsproject.org/projects/pyramid/en/latest/`.
[6]Available at: `http://flask.pocoo.org/`.
[7]IT oriented ask and answer website

+ lightweight, flexible and simple
+ less out-of-the-box approach, so the programmer has more control over the application
+ programmer chooses how to store data
+ *Jinja2*[8] templates support for dynamic web pages
+ on average [10] less verbose than Django

As a framework, Flask also brings another useful technique for web development: **routing**. Routing helps the developer to remember URLs composition in his application. On code snippet 4.1, there is an example of routing, where `route()` decorator binds `/hello` URL rule to a function `hello_world()`, thus when a user visits `http://localhost:5000/hello/`[9], in other words a client requests this page, the function is rendered in a browser.

```
@app.route("hello")
def hello_world():
   return "hello world"
```

Listing 4.1: Example of Flask endpoint.

Flask is based on *Werkzeug toolkit*[10] that implements many functions including client's request and server's response object. Werkzeug toolkit adheres to recognised Python web application development standard *WSGI*[11]. The last thing to be considered in scope of framework is its integration with ORM with a toolkit *SQLAlchemy*[12]. Instead of writing an abundance of SQL statements to perform operations over a database, SQLAlchemy loads the database into Python OOP schema, therefore each object references its respective database table. In example 4.2, the object `Animals` inherits from base class for all Flask models `db.Model` and loads animal data into its variables. If a query to retrieve data from database is executed, each animal object will contain animal data. The query to load all animal objects can use framework function `Animals.query.all()` that is equivalent to SQL query `SELECT * FROM 'animals'`.

```
db = SQLAlchemy(app) # initialise SQLAlchemy as db object
class Animals(db.Model): # initiate Animals object
   id = db.Column('animal_id', db.Integer, primary_key = True)
   fullname = db.Column(db.String(128))

if __name__ == '__main__':
   db.create_all()
```

Listing 4.2: Flask example.

Flask has support for a range of DBMS including *MySQL* and *SQL Lite*. For the purposes of the project, MySQL was chosen as it possesses scalability and a wide range of features.

[8] *Jinga2* is web templating engine for Flask.
[9] Flask uses port 5000 by default.
[10] Available at: http://werkzeug.pocoo.org/.
[11] `Python Web Server Gateway Interface`; available at: https://www.python.org/dev/peps/pep-0333/.
[12] Available at: https://www.sqlalchemy.org/.

# Chapter 5

# Design of Application

The aim of this chapter is to present the application design from different aspects. First, the application architecture is presented in connection to general communication within, followed by an aspect of data storage in section 5.2. Section 5.3 discusses the application scalability and shows the design of application class diagram and an example of general communication amongst objects. Section 5.4 sets basic rules for the test library as the main focus of the application. Section 5.5 familiarises the reader with the BDD framework JBehave and its design that is integrated into TestBuDDy. Section 5.6 focuses on test run management with the perspective of CI/CD tool. Section 5.6.1 addresses the course of action in case of emerged problem with conflicting changes in the project (SUT repository). The system design adheres to final requirements from section 3.3.3. Finally, reporting and incident management designs are presented and explained.

## 5.1 Architecture and Communication

The application schema is a simplification of MVC schema in 4.1 in order to sum up chosen technologies within the application architecture. It has been decided to refer to the application under the name **TestBuDDy** for its BDD focus and orientation on the user. As stated, the chosen backend technology is a microframework Flask. The application is implemented as a service that is ready to be connected to a frontend in the future, as specified in requirement NFR-03. Flask communicates with its *MySQL* database through ORM and toolkit *SQLAlchemy*. Section 5.1.1 is dedicated to *REST*, here is a general scenario as an example: client (browser) makes a HTTP request of type GET to view an image. Request is expected in a *RESTful* format on server side. Flask polls for the resource saved in the database through SQLAlchemy and if the image is successfully found, the backend returns the image object in its response to the client. In case that the data was not found, the server returns a negative response with an error code. The whole schema is displayed in figure 5.1.

### 5.1.1 REST API Design

[17, Chapter 19] states, that the **REST (REpresentational State Transfer)** is an architectural style of web services based on the HTTP protocol. RESTful web services, or *REST API*, recognise the main term: a resource. A resource is represented by a *URI*[1]. The

[1] *Universal Resource Identifier* is as string of characters that represent a resource.

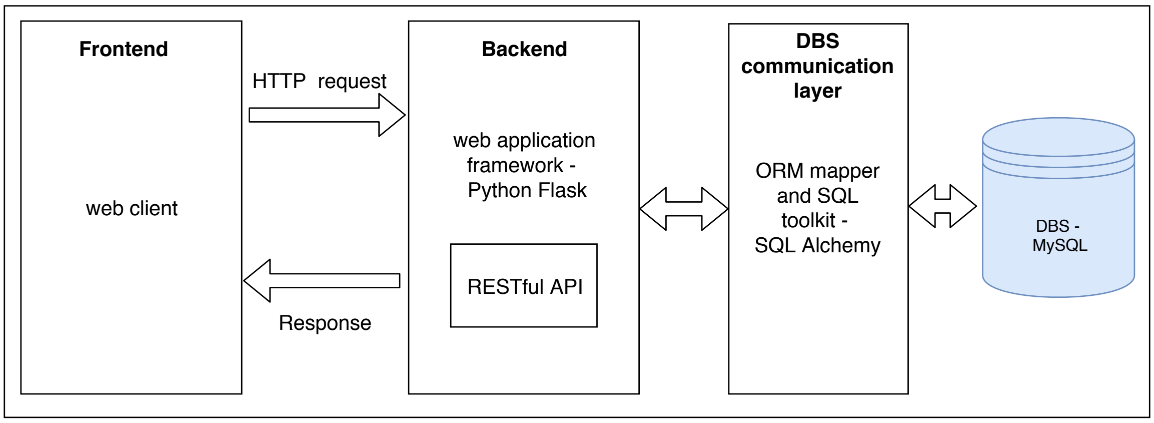


Figure 5.1: Application model.

client sends a request to this URI by means of a HTTP method (GET,POST, etc.) and the state of the resource can change. The created REST API is also further discussed in implementation section 6.2.

## 5.2 Database Design

The relational database was modelled with regards to future expansions of the application based on requirements NFR-01 and NFR-02, thus more complex database schema are in the form of an ER diagram[2], as shown in appendix C. The ER diagram of the complete database model in figure C.3 can be split into several inter-connected groups:

- user management and project management (FR-01, FR-02) - in figure C.1.
- requirement management (FR-16, FR-17) - also in figure C.1.
- test reporting (FR-11, FR-15) and incident management (FR-09, FR-10); in figure C.2.
- test design, test execution and test planning (FR-04 - FR-08, FR-11, FR-13, FR-14) - also in figure C.2.

The database section that is worth mentioning is displayed on figure 5.2. This section consists of test step table and its step definition table. Test step is based on its generated step definition. The test step definition is designed as a ”blueprint” base for the concrete step in test case. Test step definitions are saved as an active library for the each project. **Test step definition only exists if there is at least one step using it.** Blueprint is then used by multiple different steps throughout the project. Test step fills in its designated parameters within its definition. A relationship is defined by a dashed line, where its cardinality is portrayed as 2 lines (crossing the dashed line) for ”one” and line and a ”fork” for ”many”.

[2] *Entity relationship diagram*

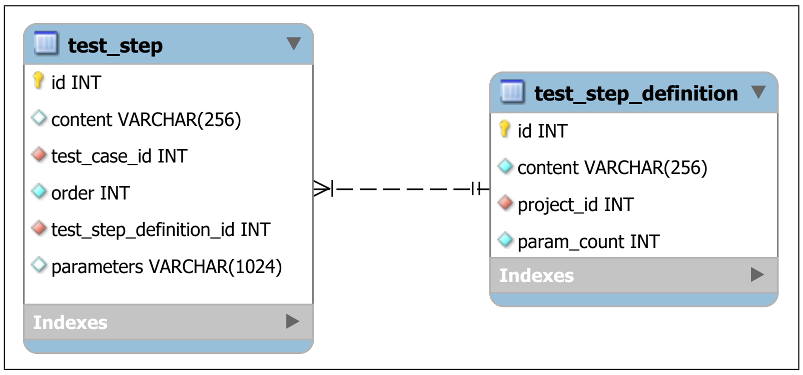


Figure 5.2: ER diagram of the database model: test step and test step definition. From the code point of view, test step definition refers to **BDD declaration** and its **BDD definition** within the BDD framework.

## 5.3 Class Diagram and Scalability

The application consists mainly of Flask *endpoints* (unique project URLs) and a main object, `Core`, used for the majority of project functionality, the exact project file division is discussed in section 6.1. Respecting the NFR-02 requirement, one of the OOP patterns used during the thesis implementation is a structural OOP pattern *Adapter*. Adapter (as stated in [11]) is an interface that binds its adaptees (the classes that inherit from this interface) to implement its methods. Client object `Core` contains concrete adaptee as its class variable and requests its methods. Thesis uses adapter pattern for:

- BDD framework, or `GherkinProcessor`,
- CI/CD tool, or `CICommunicator`,
- `IssueTracker`.

Figure 5.2 illustrates how `Core` communicates with its adapters and facilitates requested actions from each endpoint. The detailed communication is further discussed in implementation section 6.5.1.

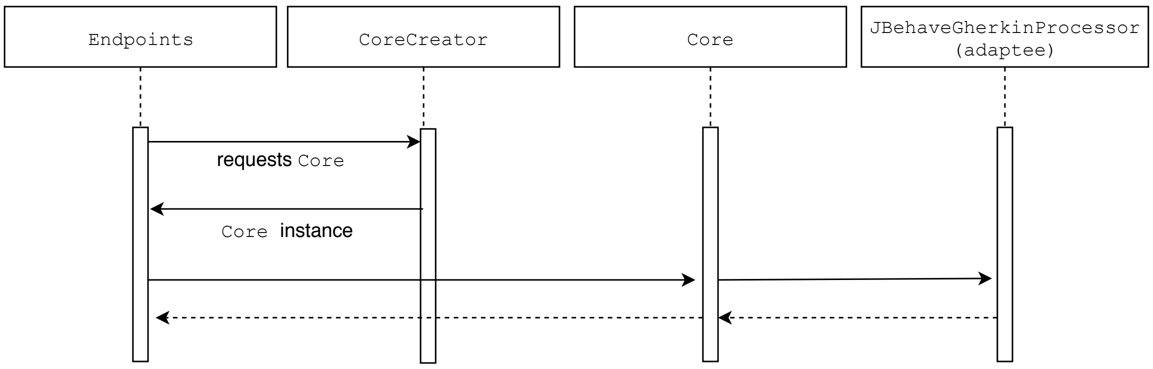


Figure 5.3: General class communication with example of BDD framework adaptee.

Another used OOP pattern is a creational OOP pattern *Factory Method*. First, based on project information (requested Adapter type), `CoreCreator` uses different factories for each Adapter and prepares a concrete adaptee for Client `Core`. Endpoint then calls for adaptee functions wrapped in `Core`. The result class diagram is in figure 5.4.

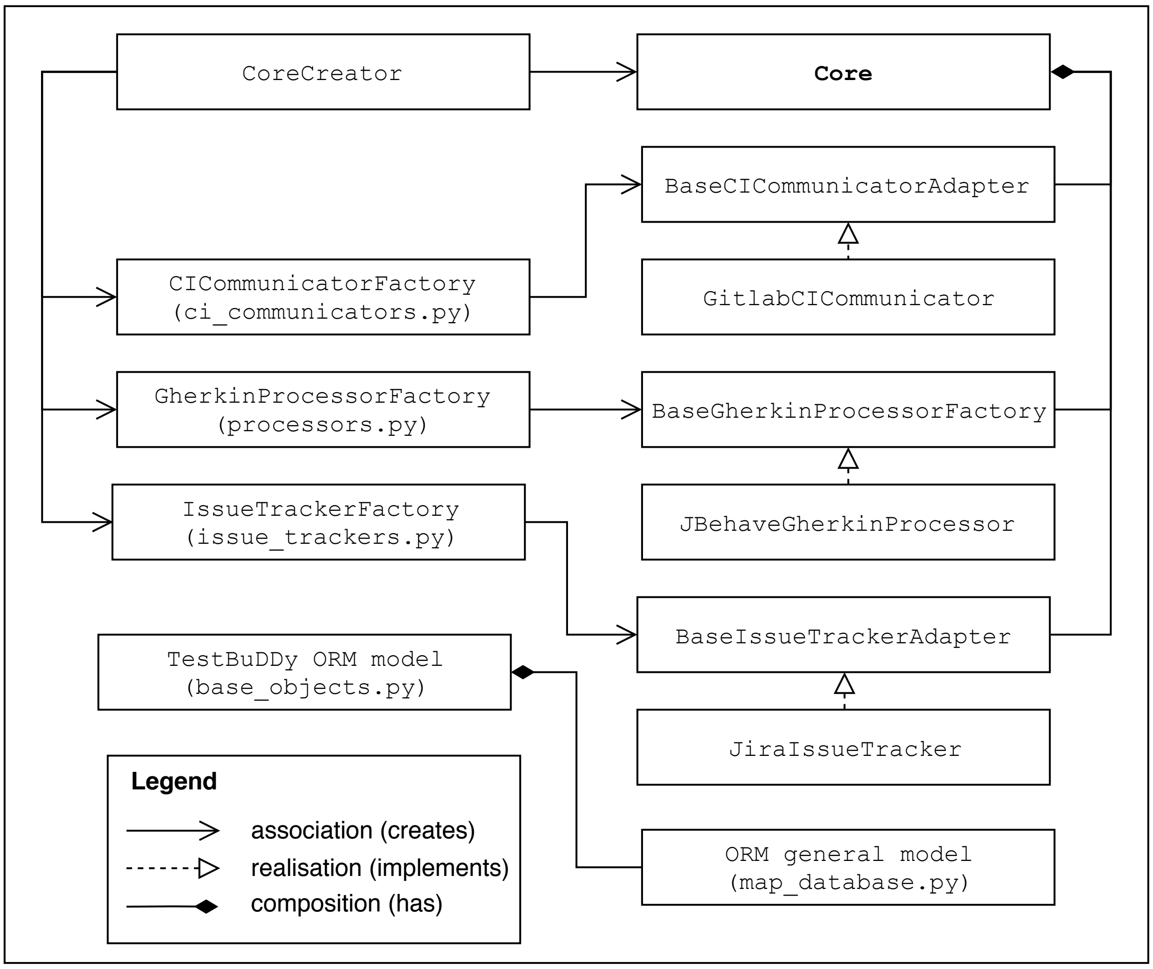

CoreCreator
Core
BaseCICommunicatorAdapter
GitlabCICommunicator
CICommunicatorFactory (ci_communicators.py)
GherkinProcessorFactory (processors.py)
BaseGherkinProcessorFactory
JBehaveGherkinProcessor
IssueTrackerFactory (issue_trackers.py)
BaseIssueTrackerAdapter
TestBuDDy ORM model (base_objects.py)
JiraIssueTracker
ORM general model (map_database.py)
Legend
association (creates)
realisation (implements)
composition (has)

Figure 5.4: TestBuDDy Class diagram.

## 5.4 Test Library

By test library, it is meant the management of projects, test plans, test modules, test cases, test steps and their test definitions. Based on FR-03, FR-04, FR-05, FR-06 and FR-07, the following figure 5.5 shows the expected flow of user actions in the application test library. These additional rules are set as:

- A user has to first connect his empty existing remote repository to a project in application[3]. This is done during project creation.
- A user then initialises TestBuDDy files[4].
- A user can proceed to manage his test library within the new project. **User actions are to be reflected on the remote repository.**

[3]Otherwise the application loses its main focus on automation (requirement FR-08).
[4]request: `<root_url>/project/<proj_id>/init_repo`

- There can be only **one test plan** connected to the project as the application presumes work with the newest version of the project code on the project remote repository.

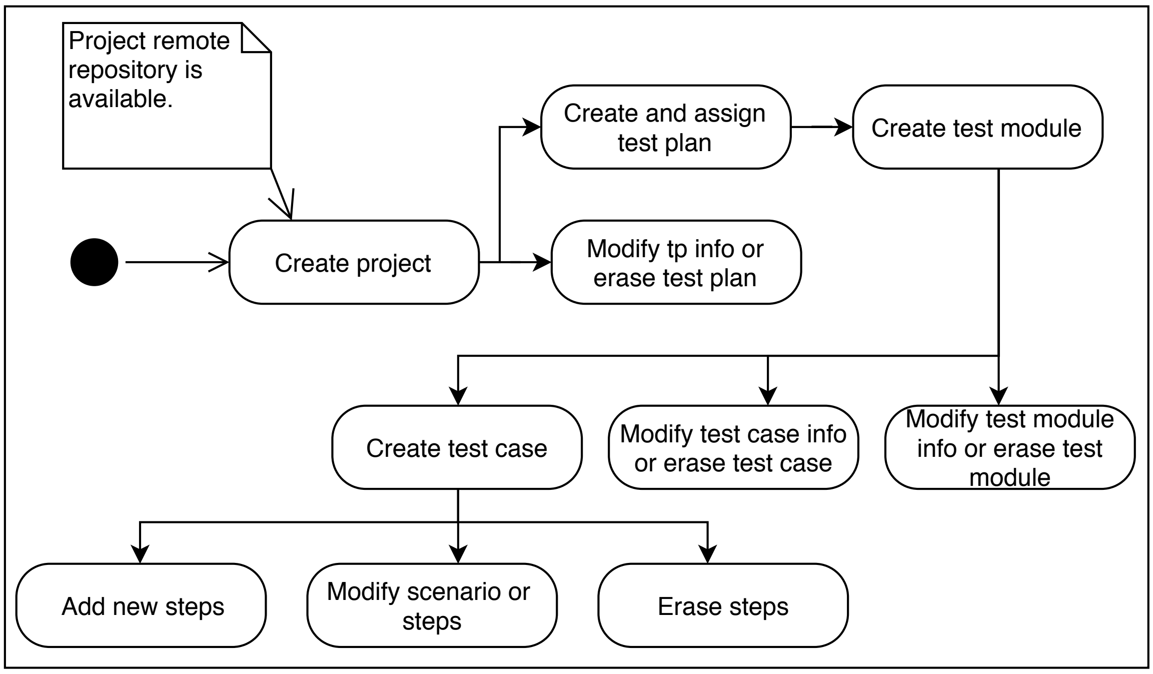

Project remote repository is available.
Create project
Create and assign test plan
Create test module
Modify tp info or erase test plan
Create test case
Modify test case info or erase test case
Modify test module info or erase test module
Add new steps
Modify scenario or steps
Erase steps

Figure 5.5: User actions within test library. Each action can be also **final**. Issues and algorithms connected with design are further investigated in section 6.5.

## 5.5 BDD Framework Integration

The thesis is implemented to work with the Java framework **JBehave**. Once the project is connected to an empty repository, a user is expected to call:

```
<root_url/projects/<project_id>/init_repo
```

to initialise the section of the repository reserved for TestBuDDy. This reserved section of the repository is divided into 2 main sections: the root of the repository, where the file `pom.xml` and a file needed for CI/CD tool[5], are initialised, and a folder `src/test` that is further divided into:

- `java`, a folder with:
  - file to run stories `TestbuddyRunnerTest.java`
  - each module step definitions under name: `<module_name>_<module_id>Steps.java`[6]. This means that module test cases share this file for implementations of their step definitions.

[5]in Gitlab CI integration case, a YAML file
[6]Module and story files are without spaces.

- `resource/stories`, with folder `<module_name>_<module_id>` for each module that contains its test cases (scenarios saved in story files) saved under names `<story_name>_<test_case_id>.story`.

The figure 5.6 illustrates this test library design.

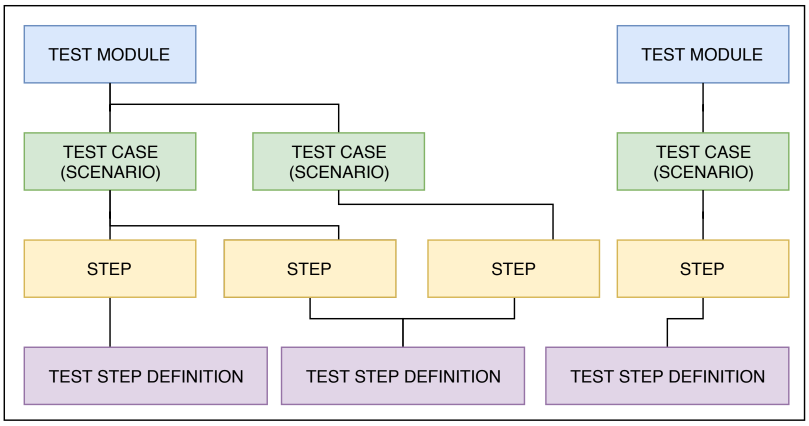


Figure 5.6: Module files consists of test step definitions of test steps in all of module test cases. All modules belong to the test plan of their project.

### Scenario Parsing

Each test case can be augmented by adding a scenario[7] with test steps. An example of such a scenario is:

```
Scenario: Logging in the application XYZ
Given User is on [logging] screen
When User uses [test@test.com] account and [test] password
Then User is redirected to [home] screen
```

Listing 5.1: Example of a test case scenario.

The scenario is then parsed and in case of new steps within the module, missing test step definitions are generated, an example of such a definition is in listing 5.4.

### Generated BDD Declarations

Test step definitions (BDD declarations and their BDD definitions) are generated surrounded by TestBuDDy footer and header comments. Users are NOT authorised to change these comment lines. A step consists of a step definition (template) and its parameters. For example the step from listing 5.2 would have the step definition[8] from listing 5.3, where `param1` is an input parameter used in the test step.

```
Then User is redirected to [home] screen
```

Listing 5.2: Example of a test step.

[7]In case of multiple scenarios within a test case, the parser accepts the first one and ignores the rest.
[8]Test step definition has its `gherkin_type` saved in connected table of the same name.

```
User is redirected to {param1} screen
```

Listing 5.3: Example of a test step definition.

An example of a generated step definition for this step is in listing 5.4, where *123* in the comments is a step definition internal unique ID.

```
/*---testbuddy---step---123---*/
@Then("User is redirected to [$param_1] screen")
public void stepCode123(@Named("param_1") String param_1) {
    /* BBD definition that is subsequently filled in by a tester locally*/
    Assert.assertTrue(true);
}
/*---testbuddy---step---123---*/
```

Listing 5.4: Generated step definition that consists of BDD declaration and BDD definition.

This JBehave design produces several issues to be resolved, this is fully described in 6.5.3.

## 5.6 Test Execution

Based on requirements FR-10, FR-11, FR-12, FR-13, FR-14, FR-15, the key functionality of the project is related to automated test runs. Test execution, or *Test Run Management*, is connected to the SUT remote repository. The remote repository consists of the tested software project and section reserved for TestBuDDy[9]. There are several scenarios to be considered:

- **Test run** is triggered automatically when BDD declarations are generated (or changed), so there is a new (or changed) test suite to be executed.
- Test run is issued by a user through TestBuDDy endpoint, this case usually involves retesting of existing test suite and will be furthermore referred as **test rerun**.
- Test run is triggered from CI/CD tool, whenever there were some changes done in the project and test repository. These changes involve supplying new BDD definitions of test steps by a user. This case also belongs to **test run** scenario, as the test suite is re-executed.

All of these scenarios involve triggering a new **pipeline** in CI/CD tool. The pipeline will be implemented as part of the Gitlab CI integration. TestBuDDy internally recognises 3 pipeline trigger types for each aforementioned scenarios: *commit* for TestBuDDy test run (TestBuDDy commited changes of the test library that triggered this pipeline), *triggered_run* for test rerun (the pipeline was triggered on an already existing commit), *other* for all other user actions executed on the remote repository. The visual demonstration of these cases is portrayed in figures 5.7 and 5.8. The possible problems of this design are discussed and resolved in the following section 5.6.1.

[9]These 2 sections are divided by folder structure.

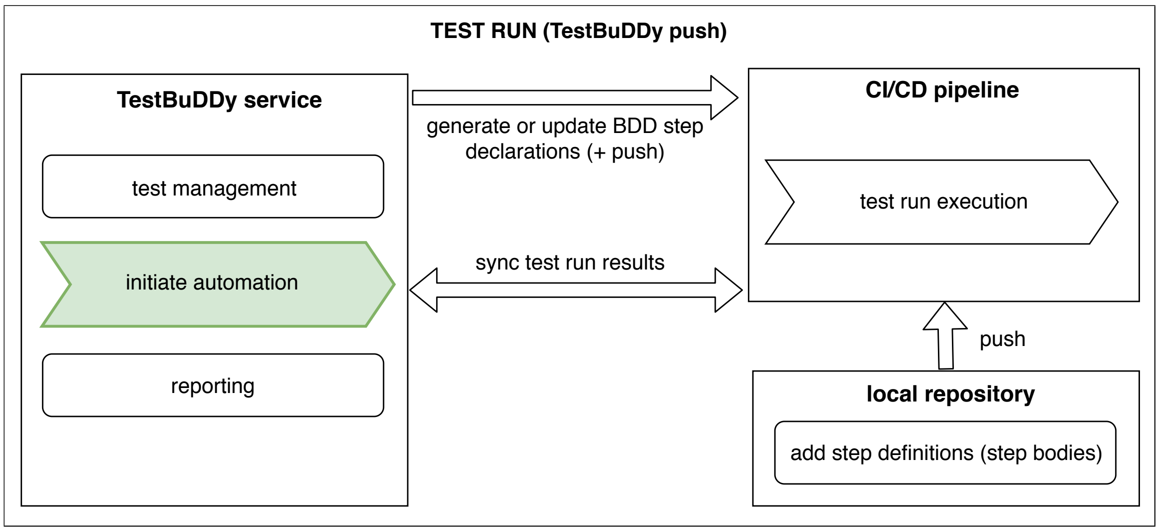


Figure 5.7: **Test run scenario**. A tester designs his test library using the TestBuDDy service. Whenever the tester adds, modifies, or removes new test steps to a test case; modifies or erases *active*[10] test cases within a test module; modifies or erases active test modules[11], it affects the test step definition library. Changed BDD step declarations are pushed onto the remote repository and executed (even in case of new BDD declarations without their definitions, which means that the test run fails as part of the Test Driven Development approach). The CI/CD tool then executes the test run as a new pipeline. When the user wants to view the result of a finished test run, the user calls the appropriate endpoint to sync and renew the test run database of TestBuDDy. The results are visible as part of the test reporting function. When the tester supplies their BDD step definitions (test steps bodies) from their local repository to the remote repository, another test run is triggered and a user can sync their test run database when in need.

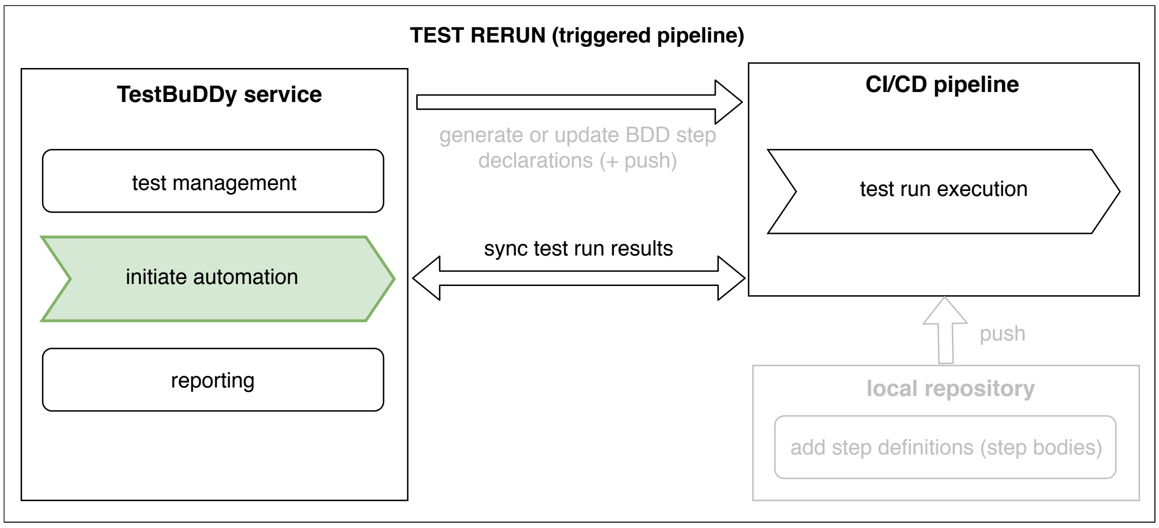


Figure 5.8: **Test rerun**. A tester can also issue a new rerun of their existing test library.

### 5.6.1 Issuing a Test Run versus New Changes

This design brings several problems to solve. What happens when there are conflicting changes done both from TestBuDDy (new or changed BDD declarations that had just been pushed into a remote repository) and changes from a local repository (changed BDD declarations or definitions during an already issued automated test run)? There are 3 possible outcomes of this scenario:

1. new local changes are ignored
2. new local changes are allowed, changes interrupt the ongoing test run and a new test run with these changes is issued
3. merge of both changes onto the remote repository and a new test rerun

In case of the 2nd and 3rd outcome, the results would have to be reported back to the service that also changes it's current test plan. Due to this unwanted functionality, **the first option is the most appropriate**. The tester is supposed to create and change the BDD step declarations within the TestBuDDy service and then fill in the BDD step definitions locally, hence when the test run is running and the tester wants to update their local changes that are now obsolete, the tester has to first **pull and merge** changes from the TestBuDDy commit(s) to their local repository. Only then, the tester can again fill in the BDD step definitions, and trigger a new test run by their manual push.

Each test run is saved in the database in the `ci_run` table for each pipeline of the project. When a user syncs their TestBuDDy test run database with the CI/CD tool, the application renews status of the now finished test runs (the pipelines that did not finish their activity are in a *pending* status). The user is expected to use an endpoint to sync with the CI/CD tool[12]. Test run library synchronisation is further discussed in implementation section 6.6.

[13] [14]

### 5.6.2 Reporting

Based on FR-15, the application has to analyse and parse results of finished pipelines. A report for each test run (pipeline) can have 3 different final results:

1. **CI build failed** - when Gitlab „build“ *job*[15] failed, so JBehave tests weren't even compiled and executed
2. **success** - all tests were executed with a successful result
3. **failed** - there is at least one test that failed

In the latter 2 cases, the report saves the results of each executed test case. This is done by parsing the *tracefile* from Gitlab CI job „test“ (TestBuDDy suggests to use „test“ job in the user's CI file). If the test case fails, the report also saves information about which step had failed, this is visible in listing 5.5 The report needs to be generated based on the finished failed or successful test run (pipeline). Reports are generated against saved test

[12] `<root_url>/projects/<project_id>/ci_runs/<pipeline_id>/sync`

[13] Active test case is a test case with test steps.

[14] Active test modules is a module with at least one active test case.

[15] A logical unit or a phase in Gitlab CI YAML file.

runs and are parsed based on the used BDD framework and the CI/CD tool. Reports can be printed for the project or one report can be viewed for a specific test run (`CI Run`)[16].

```
"result": "failed",
"test_cases": {
  "stories/LoginModule_42/LoggingintheapplicationXYZ123.story": {
    "failed_at_step": "Given User is on [logging] screen ",
    "result": "failed"
  },
  "stories/WeatherMod_6/UglyDay1.story": {
    "result": "success"
  },
  "stories/WeatherMod_6/SunnyDay4.story": {
    "failed_at_step": "When Show [temperature] for [Sunday] option is [
       disabled]",
    "result": "failed"
  }
}
```

Listing 5.5: Example of failed test run report body with executed test cases (story files).

### 5.6.3 Incident Management

Based on FR-09, SUT (*Software under test*) bugs are automatically generated from reports. For TestBuDDy, a bug is a test step (test step definition with certain input parameters) broken in at least one or more test cases. Bugs are also printed out with reference to the project specific test run (requested by its pipeline identifier in endpoint) or for the whole project. An example of a bug is in the following listing 5.6.

```
"ci_runs_affected": [
  {
    "id": 34,
    "pipeline_id": 1791
  },
  {
    "id": 33,
    "pipeline_id": 1790
  }
],
"data": {
  "broken_step": "Given User is on [home] screen ",
  "test_cases_affected": [
    "stories/LoginModule_42/Loggingintheapplication123.story",
    "stories/LoginModule_42/RolesLogin12.story",
    "stories/BasicFunctions_66/ApplicationSmokeTest345.story"
  ]
},
"id": 2,
```

[16]At the endpoint level, a CI Run is always referred to by its pipeline ID within project for users convenience.

```
"link": "SUT-123",
"status": "open"
```

Listing 5.6: Example of bug body.

## 5.7 GUI Design for Future Expansions

Although the application is designed as a service, the service is meant to be used with a frontend GUI in the future. The application GUI design is inspired by the BDD orientated tool *Hiptest* and the test management tool *Testlink*. The GUI is organised based on the requirement groups from section 3.3. The wireframe example of the GUI is in figure 5.9. The UI is split into:

1. **menu** (left side), that also contains application **logo** on top. Menu categories are based on set requirements and each menu option opens up its respective screen with content.

2. **main content** (rest), which is a screen that will always contain the **user management** section on top. The user management section contains information on who is logged in, log out button, etc.

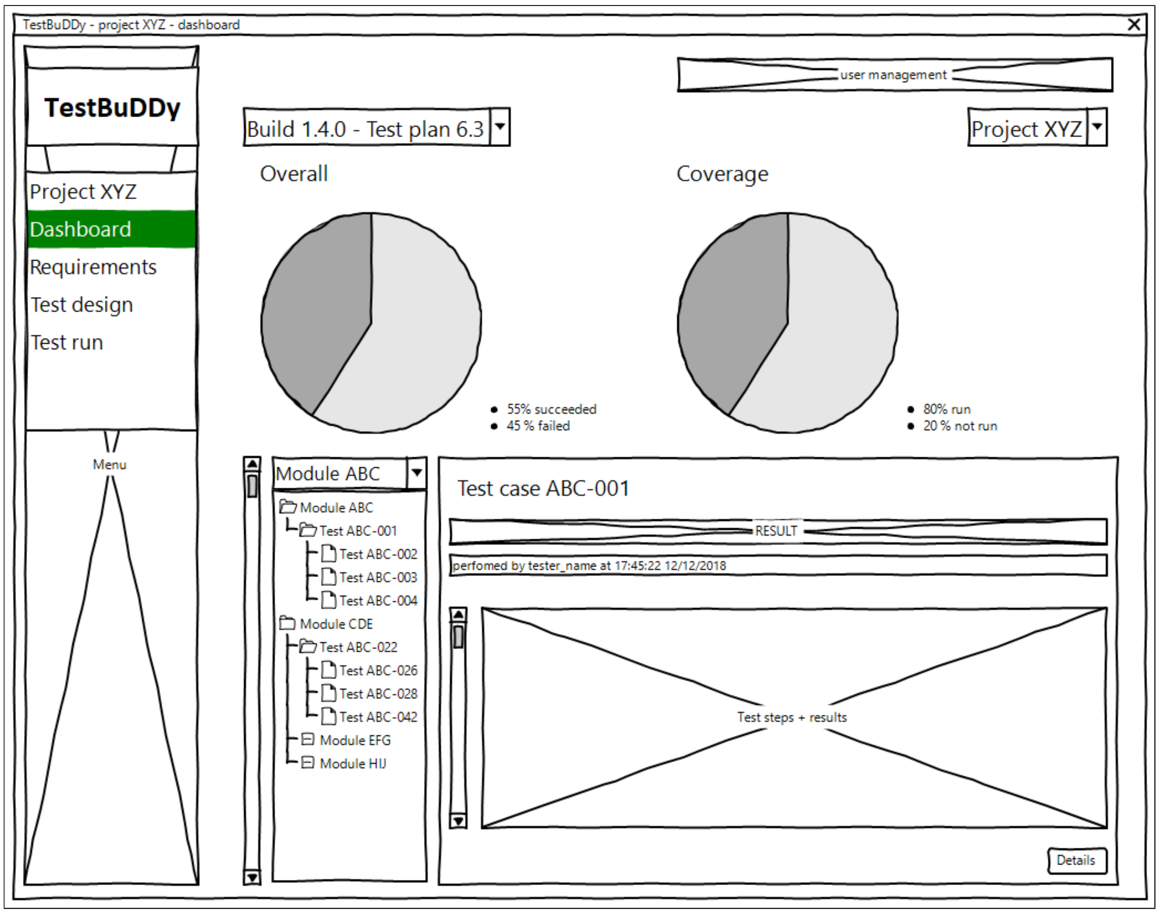


Figure 5.9: Wireframe of GUI exemplary design for the *TestBuDDy* application: dashboard. Dashboard screen main content shows test reports with regards to test cases and a test plan.

The TestBuDDy endpoints are designed using HTTP methods *GET*, *POST*, *PUT* and *DELETE*. Modern browsers only support sending GET and POST type requests, therefore in order to fully connect a GUI frontend with the TestBuDDy, it is recommended to use *Javascript*[17] scripting language. The reason for this choice is that the popularity of Javascript grew until the point, where its client-side version is used on over 95% of web pages since 2018 [4].

[17]Available at: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference`.

# Chapter 6

# Implementation Details

This chapter mainly serves to provide detailed information regarding interesting or crucial parts of a TestBuDDy implementation. Sections 6.1 and 6.2 provide general project implementation details about application file structure and REST API endpoints. The remaining sections address either set requirements from section 3.3, or discuss specific issues that became apparent during design implementation and needed to be further addressed, or closely looked into current TestBuDDy integrations in context of test run management, reporting and incident management functionalities. Section 6.5 is fully dedicated to design issues with the test library, specifically what happens when simultaneous test step changes are sent to a TestBuDDy test case (in form of a scenario), removal of a test module, or a test case, etc. Section 6.6 pays attention to the test run library during synchronisation between the CI/CD tool and the application. Section 6.7 closes up the implementation chapter with few details regarding its JIRA integration. This chapter also summarises available functions that a user can do within the application. Each action is atomically represented by its responsible endpoint. The complete documentation of each endpoint is visually available via the *Swagger* tool that is discussed further in section 6.2. On top of the corresponding source code, there are also comments with requirement tags for each covered requirement from table 3.2 (example of such comment within a source code is in listing 6.1).

```
# TestBuDDy-Requirements-coverage
# NFR-01, NFR-02
```

Listing 6.1: Example of requirements coverage comment.

## 6.1 Application Schema and File Structure

The application is run and deployed by the container platform *Docker*[1]. Docker helped to ship and connect all required application dependencies and database. There are 2 docker containers: `backend`, where the application functionality resides, and the `database`, where the database is deployed[2]. Schema in figure 6.1 portrays both containers and overall project file structure. Docker commands to deploy TestBuDDy can be found in appendix B (manual). File `run.py` is an executable file that runs the TestBuDDy application.

[1] Available at: https://www.docker.com/.

[2] There is also a third container `unittest` that is reserved for unit tests. Section 7 desribes this in further detail.

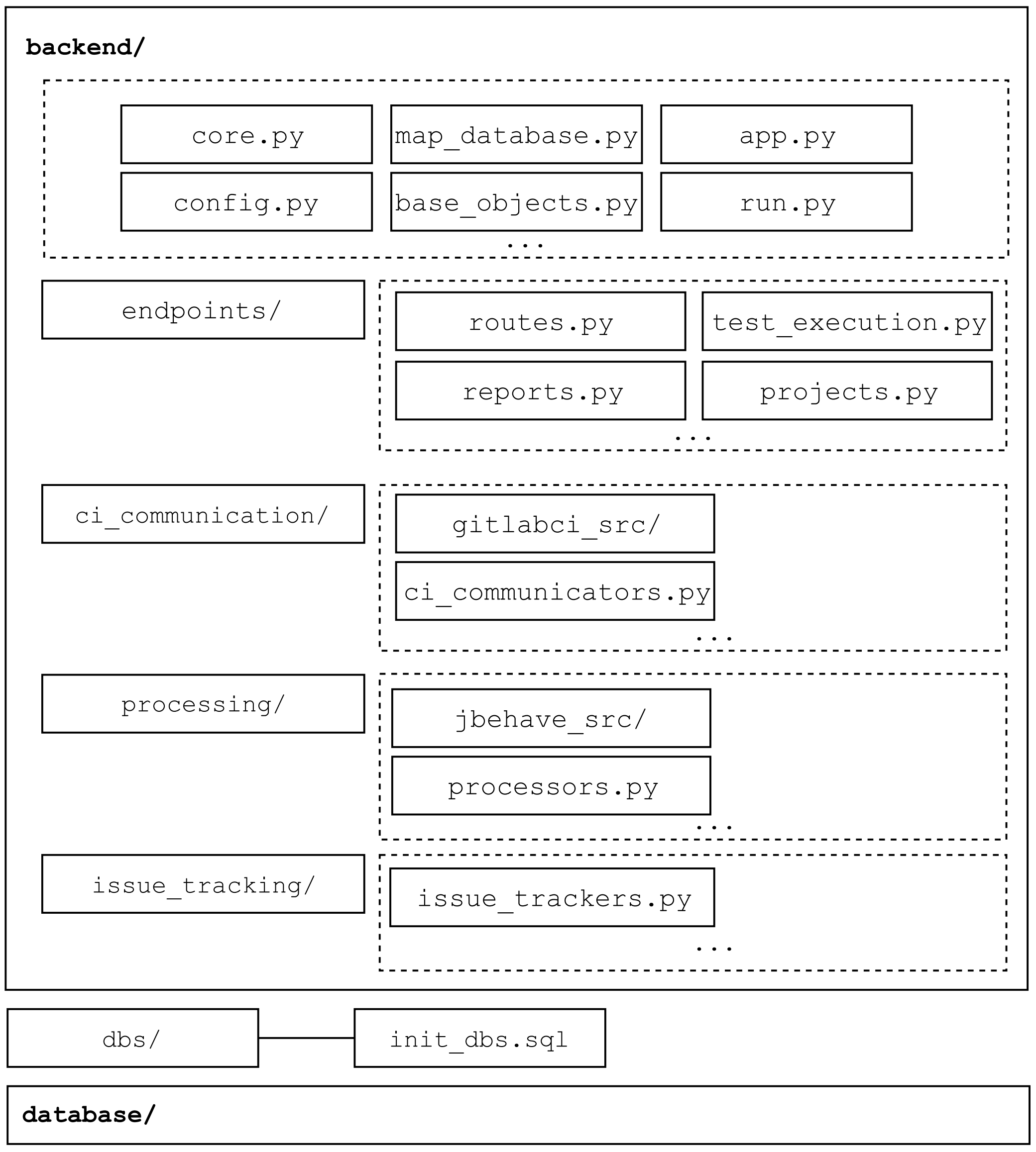


Figure 6.1: TestBuDDy project file structure. Files are divided into folder groups. Containers are shown in bold. There are multiple backend subfolders: `endpoints` contains all endpoints, `ci_communication` contains the CI/CD tool connection[3], `processor` is related to BDD framework and BDD language processing connection and `issue_tracking` contains issue tracker connection.

## 6.2 REST API Endpoints

TestBuDDy is a web service that provides its functionality and communicates with a user through specific *endpoints* (specific URL addresses).

[3]TestBuDDy also refers to `processor` as *language processor* or *gherkin processor*.

### 6.2.1 Endpoint Design

REST API endpoints are designed in plural form. Each endpoint is meant to fully represent each resource tree, thus it is explicitly required to fill in all needed valid source identifications. For example, by calling the following endpoint which accepts a *GET* request:

`<root_url>/projects/<project_id>/plans/<testplan_id>`

an existing test plan with a set ID belonging to an existing project with a set ID is shown, otherwise an error code and an error message is returned in response. A real example of this endpoint could then look like this:

`127.0.0.1:5000/projects/1/plans/16`

where the project with an id of 1 is supposed to have its assigned test plan with and id of 16.

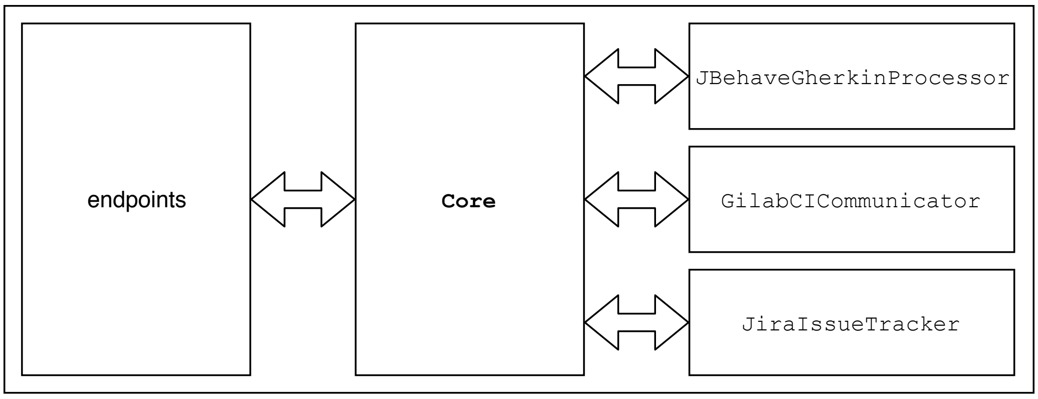


Figure 6.2: Example of communication amongst endpoints, `Core` and its class object instances for chosen integrations.

### 6.2.2 Response

In case of a negative response, an error code and an error message `/text/html` are returned. The error message can also contain specific *exception* information which can be helpful to users[4]. In case of a positive response, a response code 200 and a JSON object is returned:

`{„message": <response message>, „object": <requested object>}.`

Returned response codes adhere to HTTP standard: 200 for a positive result of the operation, 500 for an internal server error, 404 for when a requested resource was not found, etc. The result summary for possible TestBuDDy responses are as follows:

- **200** OK; requested resource returned,
- **400** NOT OK; insufficient request body parameters,
- **404** NOT OK; resource not found or the parent resources are not valid[5],
- **415** NOT OK; unsupported media type,
- **500** NOT OK; other TestBuDDy error.

[4]TestBuDDy own general exception is called `RuntimeError`.
[5]For example when requesting a test plan, its parent project was not found under its provided path ID.

### 6.2.3 Swagger Doc

The detailed REST API for the whole project is documented with the *Swagger* tool[6] and visible by calling the root endpoint (`<root_url>`[7]). Endpoints were divided amongst several categories, based on the groups from the project requirements table 3.2. Examples of the Swagger screen are shown in figures D.1 and D.2. As mentioned in section 5.3, the `Core` object communicates with its instance objects upon the request call. An example of this communication with current TestBuDDy integrations is located in figure 6.2.

### 6.2.4 SQL Alchemy and Endpoints

Endpoints are designed to exclusively commit persistent changes into the database. An endpoint always opens a new *session* for database transactions, then uses `Core` to implement requested endpoint functionality and finally commits all database changes that were up to this point only pending transactions (`Core` and all of its parts are only *flushing* pending changes). By default, `session.flush` is part of the `session.commit` command as part of the `autoflush` option in settings, but it was possible to fully divide these operations and gain full control over which changes were committed and which were not. This approach was mainly used because of the possible *rollback* function in case any errors had occurred during TestBuDDy actions. Rollback is used to annul executed flushes, protecting the database from any discrepancies between the test library in the repository and the state of the repository.

## 6.3 User and Project Management

A user is able to manage and view users within the system, as discussed in part 3.3. User management is created to be general, as TestBuDDy is designed to be run mainly as a service. Nevertheless for future scalability, the database and endpoints were designed to deliver role support, where role management is provided and a user can have a role, or a test case assigned, or the assignment can be removed, etc. These functions are executed when calling their appropriate endpoints (please see further details in Swagger doc, sections *User and role management* and Project management).

## 6.4 Requirement Management

A user can manage, tag and view their requirements as mentioned in chapter 3.3. The user can also create, view and manage their project tags. These functions are executed when calling their appropriate endpoints (please see further details in Swagger doc, section *Requirement management*).

## 6.5 Test Library

Once a user has connected their project with their remote repository, the user can proceed with their test library management. Test library management involves test specification and test execution. The design of the test library from 5.4 highlighted several issues to resolve.

[6]Available at: `https://swagger.io/`.
[7]By default, the `root_url` consists of a localhost address and the default Flask port (5000).

### 6.5.1 Test Step Management

One of the core TesBuDDy functionalities is generation of test step definitions when a user simultaneously adds, modifies or erases test steps within an existing test case. Although step management actions are all atomic within the test case, the user wants to send the whole scenario with all of these possible actions for each step in the test case. A scenario is sent whole (an example of such a scenario was already shown in figure 5.1) for the users convenience. The whole process of test step management within the test case is summarised in diagram 6.3. The diagram shows all actions that the backend has to execute, from when the user supplies a new or changed scenario with test steps until a request response (the response is not portrayed in the figure, as the figure is mainly focused on backend actions).

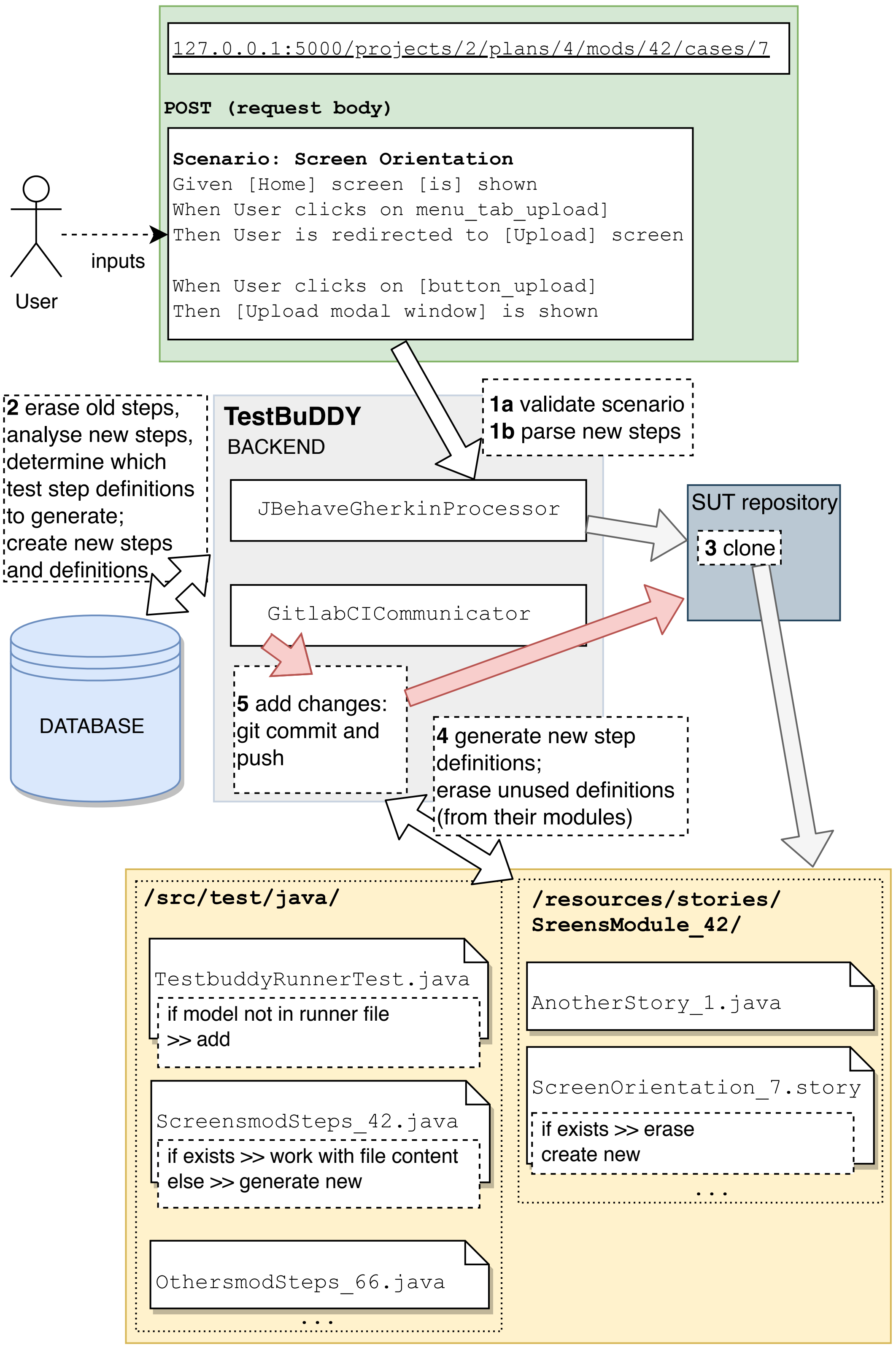
The question is, how to manage all of the step actions done within one test case scenario at the same time? Imagine when the user edits the test case scenario, the first phase is to analyse new test step definitions used within the scenario to uncover any duplicates with the current test step definitions. The second phase is to analyse the test steps themselves. There are 2 possible resolutions to solve the test step analysis:

1. Analyse all current (now old) test case steps. If the step was:
    - **modified**, find all steps that still use its step definition and if there are any, generate a new step definition, otherwise modify the step and its step definition (in the database and on the repository)
    - **erased**, find all steps that still use this step definition and if there are none, erase this step and its definition (similar to algorithm 2).
2. Atomic approach: Erase all current (now old) test case steps on the database layer and create all new steps according to algorithm 1 (in this section), this includes reuse of current test step definitions and final cleanup of now unused step definitions in the database and on the repository.

**Algorithm 1:** Creating/modifying test steps.

```
Result: new test step and its definition processed
1  foreach new test step do
2      try to find its test step definition (from current step definition database)
3      if test step definition found then
4          create test step
5          assign found step definition to step
6      else
7          create test step definition
8          create test step
9          assign created step definition to step
10     update changes on database
```

**The second option** was evaluated as more appropriate, not only because it follows the atomicity of TestBuDDy actions throughout the service, but also because it is an easier solution to implement with the desired effect on TestBuDDy functionality.



127.0.0.1:5000/projects/2/plans/4/mods/42/cases/7
POST (request body)
Scenario: Screen Orientation
Given [Home] screen [is] shown
When User clicks on menu_tab_upload]
Then User is redirected to [Upload] screen
When User clicks on [button_upload]
Then [Upload modal window] is shown
inputs
User
2 erase old steps, analyse new steps, determine which test step definitions to generate; create new steps and definitions
TestBuDDY BACKEND
1a validate scenario
1b parse new steps
JBehaveGherkinProcessor
GitlabCICommunicator
SUT repository
3 clone
DATABASE
5 add changes: git commit and push
4 generate new step definitions; erase unused definitions (from their modules)
/src/test/java/
TestbuddyRunnerTest.java
if model not in runner file >> add
ScreensmodSteps_42.java
if exists >> work with file content else >> generate new
OthersmodSteps_66.java
...
/resources/stories/ SreensModule_42/
AnotherStory_1.java
ScreenOrientation_7.story
if exists >> erase create new
...

Figure 6.3: Diagram describes the whole process when a user adds/modifies/removes steps within an existing test case. A POST request with a scenario is sent. Each dashed box represents a TestBuDDy action that is carried away in sequence that is marked with bold numbers.

In case of a scenario modification, the user can first view (or copy) the current test case scenario (if any was provided) by calling the appropriate endpoint to view their test case information. User first calls the following endpoint of a GET type:

```
<root_url>/projects/<proj_id>/plans/<tp_id>/mods/<tm_id>/cases/<tc_id>
```

The request needs its full path of correct identifiers for the project `proj_id`, the test plan `tp_id` and the test case `tc_id`. Then the user can proceed with step processing and endpoint generation as shown in diagram in figure 6.3. The endpoint did not change, but the request type, POST, differs and the user sends his scenario in the body of the request.

The `Core` object is requested from the endpoint which uses its instance objects for the communication with the CI/CD tool, the BDD framework and the issue tracker. The user sends the whole scenario in the request body as mentioned above in this section. An example of communication during step modification of the test case is in figure 6.4.

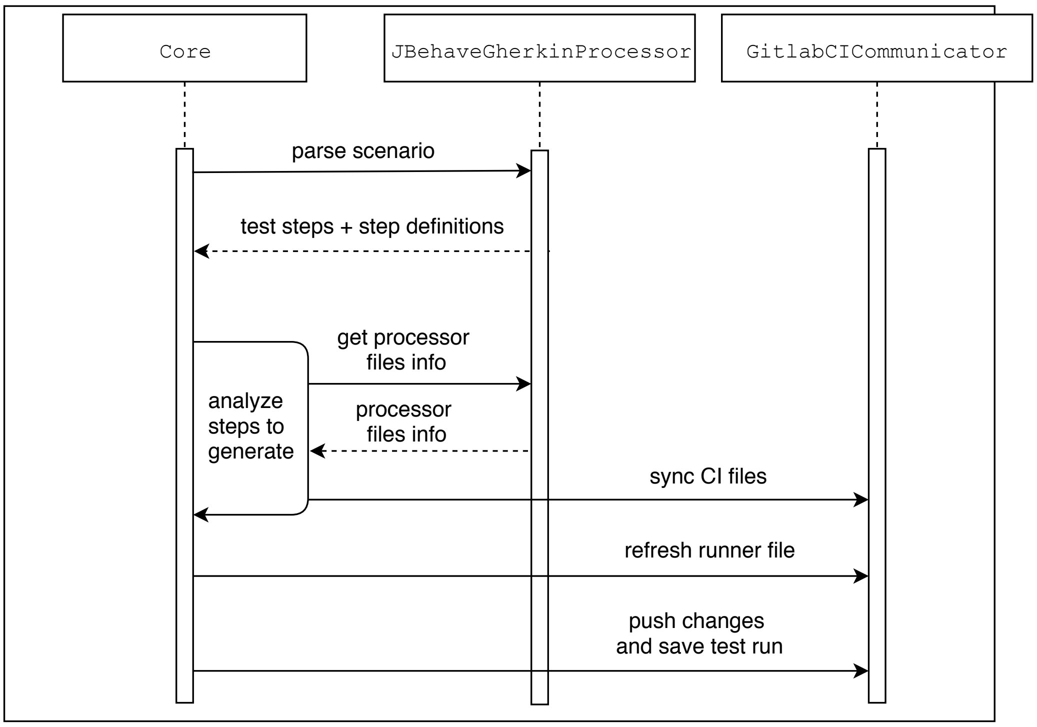


Figure 6.4: Communication diagram when a user changes test steps in the test case[8].

### 6.5.2 Test Step Definitions Library

The design also implied that a user will want to see their current test step definitions in use. For this purpose, the user is able to view their current test step definition library as requested by FR-06. The test step definition can be viewed with respect to the project or the test definition unique identifier. This functionality is executed when calling the appropriate endpoint (please see full Swagger doc for further details, section *Test library - cases and steps*).

[8]This includes adding new test steps, modifying them or remove within test case scenario.

### 6.5.3 Test Case and Test Module Removal

As mentioned before, test case steps consist of step declarations and arguments. Each declaration has its own definition that contains source code of a step definition. As discussed in 5.5 the source code of each test step definition is saved within a test module file whose test case first used this test step definition. In other words, module test cases share one module file, as visible in figure 5.6 from previous chapter. One of the addressed design issues is removal of an existing *active*[9] test case. Algorithm 2 describes actions when removing a test case.

**Algorithm 2:** Removing a test case and its steps from a test module

```
Result: test case, its steps and module unused test step definitions removed
1 remove story file
2 erase all its test steps (update database)
3 get all test step definitions of test module (that test case belongs to)
4 foreach test step definition (and its gherkin type) do
5     foreach test module in test plan (that test case belongs to) do
6         try to find test step definition
7         if test step definition found then
8             erase test step definition[10](from module file)
9     erase test step definition (update database)
```

Alternatively, when creating a new test step, TestBuDDy always looks whether its definition is not already in the project (in another module), thus a test step definition does not need to be generated. Another issue shown in algorithm 3 is the removal of a test module and all of its active test cases.

**Algorithm 3:** Removing a test module

```
Result: test module removed and all still used test step definitions moved to other
        module
1 foreach test case in module do
2     remove test case as in algorithm 2
3 get all remaining test step definitions of test plan (that test module belongs to)
4 foreach test step definition do
5     find test step definition in module file
6     if test step definition found then
7         move test step definition from module file to first found active module in test
          plan
8 remove module from runner file
```

[9]Test case with test steps.

[10]a test step definition = a BDD declaration and a BDD definition; surrounded by a header and a footer comment with a test step definition unique ID.

## 6.6 Test Run Management and Reporting

A user can view currently saved test runs by calling the appropriate endpoint:

`<root_url>/projects/<project_id>/ci_runs`

This endpoint only returns a list of test runs in the database. To have the list of test runs refreshed, a synchronisation with the CI/CD tool needs to be called. As indicated in section 5.6.1, the user is expected to use the endpoint to synchronise with the used CI/CD tool's current data using the following endpoint of a POST request type:

`<root_url>/projects/<project_id>/ci_runs/<pipeline_id>/sync`

This request synchronises the TestBuDDy database with current pipelines and their states within the CI/CD tool, then creates reports and in case of a failed report generates and refreshes found bugs and synchronises them with the connected issue tracker (if any). A default request setup is to synchronise all, nevertheless the user can modify its request query parameters for selective synchronisation. The communication diagram in 6.5 describes the full process of ongoing synchronisation. During the implementation of reporting, it was discovered that reports always need to be evaluated and when required refreshed against current data in the CI/CD tool in order to always have a valid report for all finished test runs. Reports are also optimised in a way that they are created only if the test run is in the finished state. The user can view currently saved test runs by calling the following endpoint:

`<root_url>/projects/<project_id>/ci_runs/reports`

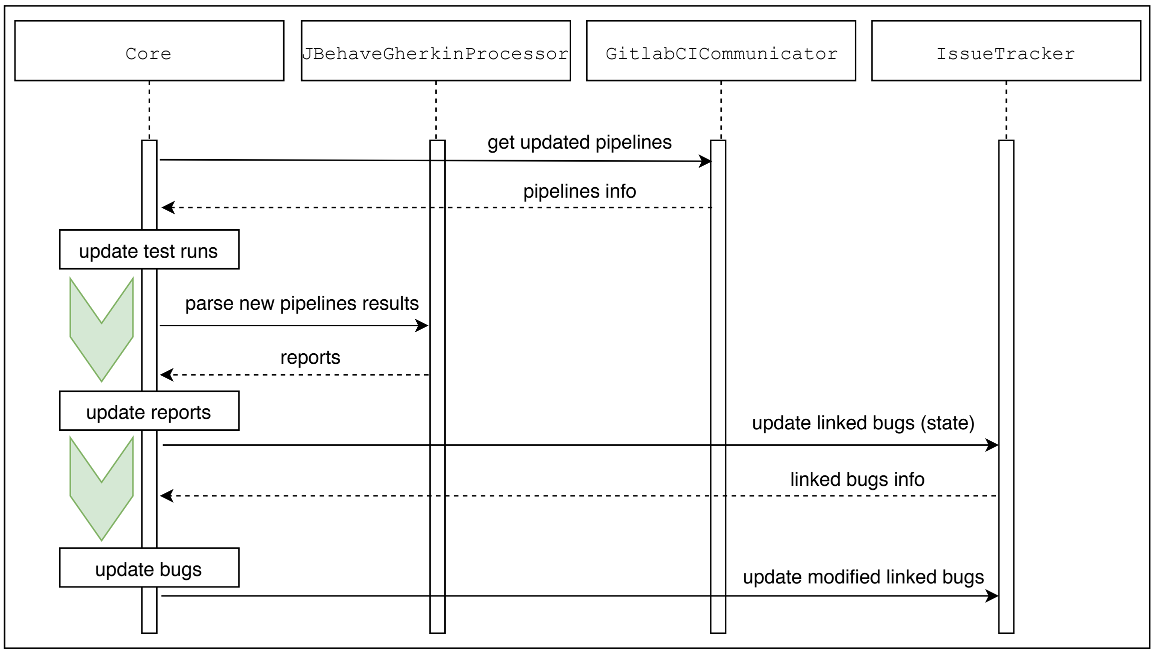
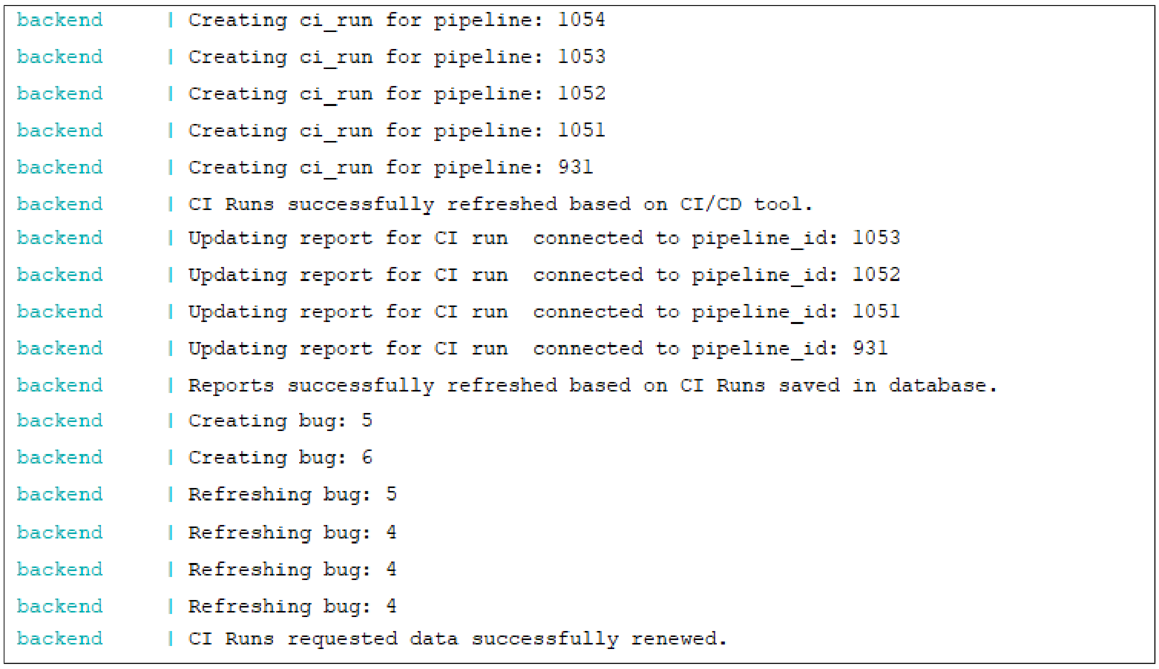


Figure 6.5: A communication diagram for the synchronisation with the CI/CD tool. Green arrows represent the action based on the dependency on the source of the arrow.

### 6.6.1 Gitlab CI Integration

Gitlab CI returns a complete list of pipelines including the ones that could have been refreshed in previous synchronisation. TestBuDDy optimises this long process: only new pipelines and the pipelines with an *unfinished state* (any other state than „success“ or „failed“) are refreshed, the rest is simply processed. Because the synchronisation can take some time, TestBuDDy also informs the user about its current operation in the service CLI[11], an example of logging output is in figure 6.6. The user can also view reports and bugs separately or in respect to the current project test run library (without synchronisation), this functionality is executed when calling each appropriate endpoint (please see full Swagger doc for further details).



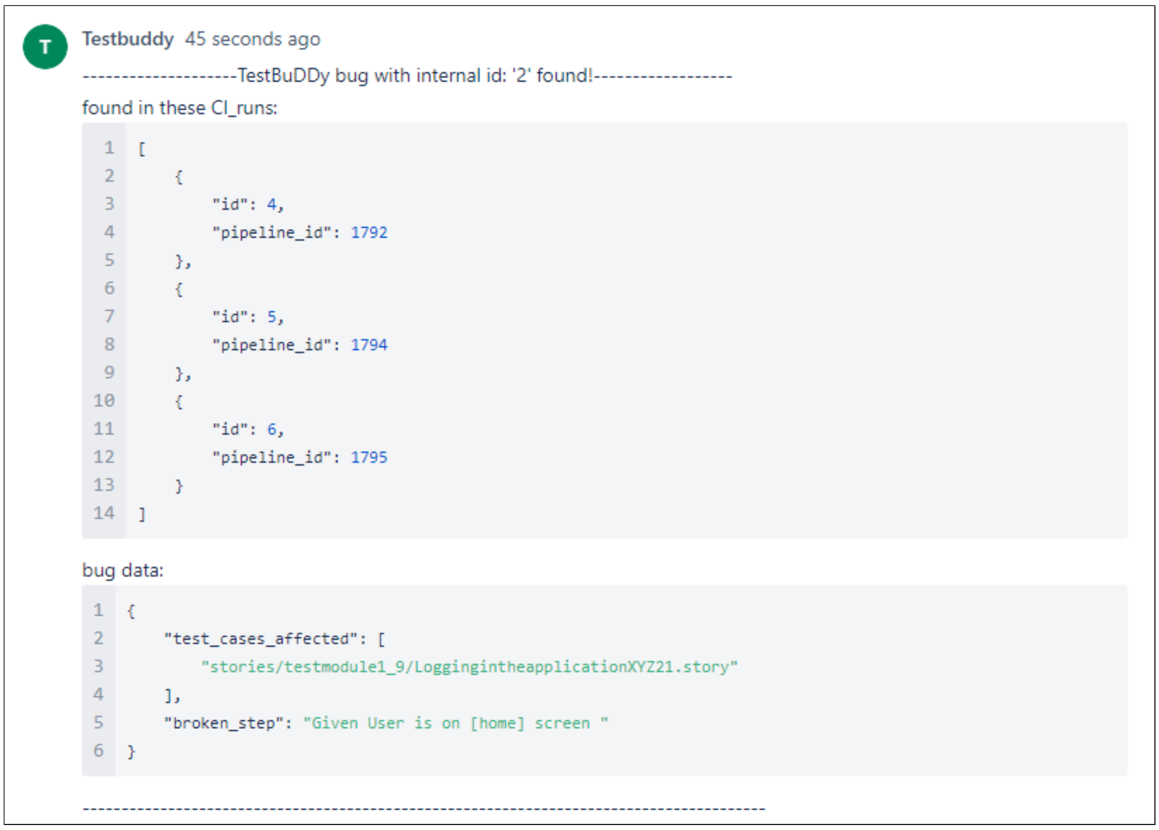
```
backend      | Creating ci_run for pipeline: 1054
backend      | Creating ci_run for pipeline: 1053
backend      | Creating ci_run for pipeline: 1052
backend      | Creating ci_run for pipeline: 1051
backend      | Creating ci_run for pipeline: 931
backend      | CI Runs successfully refreshed based on CI/CD tool.
backend      | Updating report for CI run  connected to pipeline_id: 1053
backend      | Updating report for CI run  connected to pipeline_id: 1052
backend      | Updating report for CI run  connected to pipeline_id: 1051
backend      | Updating report for CI run  connected to pipeline_id: 931
backend      | Reports successfully refreshed based on CI Runs saved in database.
backend      | Creating bug: 5
backend      | Creating bug: 6
backend      | Refreshing bug: 5
backend      | Refreshing bug: 4
backend      | Refreshing bug: 4
backend      | Refreshing bug: 4
backend      | CI Runs requested data successfully renewed.
```

Figure 6.6: Example of TestBuDDy logging during synchronisation with the CI/CD tool. Full refresh of test management library including reports and bug generation.

## 6.7 Incident Management: JIRA Integration

Issues, or bugs, are automatically generated with regards to the specific broken test step. Each issue keeps information about the test cases which were affected by this issue, which means they failed. Bugs are always generated based on current reports in the database (synchronisation can be issued as part of a synchronisation request from the previous section). Based on the requirement FR-10, each bug can be provided with a link to an existing bug within the issue tracker to allow *cross referencing*. When the issues are synchronised, the cross referencing with an issue tracker is called to update bug related information (affected pipelines) on the tracker and vice versa. TestBuDDy also updates the status of each linked bug based on its current JIRA data. Cross referencing in JIRA integration is done by adding bug reports in the form of a comment on the linked issue tracker bug. The whole process of issue synchronisation updates on both sides is triggered as part of test run management synchronisation from the previous section and can also be triggered selectively based on the current test run library (also as part of the same synchronisation endpoint from the previous section). An example bug report in JIRA GUI is in figure 6.7

[11] *Command line interface*


Testbuddy 45 seconds ago

--------------------TestBuDDy bug with internal id: '2' found!------------------

found in these CI_runs:

```
[
    {
        "id": 4,
        "pipeline_id": 1792
    },
    {
        "id": 5,
        "pipeline_id": 1794
    },
    {
        "id": 6,
        "pipeline_id": 1795
    }
]
```

bug data:

```
{
    "test_cases_affected": [
        "stories/testmodule1_9/LoggingintheapplicationXYZ21.story"
    ],
    "broken_step": "Given User is on [home] screen "
}
```

------------------------------------------------------------------------------------------

Figure 6.7: TestBuDDy generated comment with current bug status when the bug library was updated on both sides (full synchronisation).

# Chapter 7

# Application Testing and Automated Test Suite

An integral part of the application development cycle was its testing phase. This phase can be split into 3 main stages:

1. **acceptance testing**, where the compliance with the application requirements is validated,
2. **automated test suite**, where all endpoints are being tested,
3. **usability testing**, where users evaluated the application.

## 7.1 Acceptance Testing

TestBuDDy has undergone the acceptance testing phase in order to ensure its compliance with the set functional requirements from table 3.2. Additionally, scenarios/actions within the application were created and connected to the requirements that each scenario covers. The results are summarised in tables 7.1 and 7.2. The acceptance testing was executed manually.

| **dependency** | **id** | **scenario** | **expected result** | **req. coverage** | **state** |
|---|---|---|---|---|---|
| | AT-1 | Create a project. Initialise its repository. | Project created. Repository initialised. | FR-02 FR-08 | OK |
| AT-1 | AT-2 | Create a test plan. Assign the test plan to the project. | Test plan created. | FR-03 FR-04 | OK |
| AT-2 | AT-3 | Create a test module. | Module created. | FR-05 | OK |
| AT-3 | AT-4 | Create a test case. | Test case created. | FR-05 | OK |
| AT-4 | AT-5 | Add, erase or modify scenario (test steps) to the test case. | Steps added. Repository changed. | FR-06 FR-07 FR-13 FR-14 | OK |
| AT-2 | AT-6 | Rename the module. | Repository changed. | FR-06 | OK |

| **depen-dency** | **id** | **scenario** | **expected result** | **req. cover-age** | **state** |
|---|---|---|---|---|---|
| AT-2 | AT-7 | Erase the module. | Repository changed. | FR-06 | OK |
| AT-3 | AT-8 | Rename the test case. | Repository changed. | FR-06 | OK |
| AT-3 | AT-9 | Erase the test case. | Repository changed. | FR-06 | OK |
| AT-4 | AT-10 | Synchronise the test run library. | Test runs synchronised. Reports and issues synchronised. Linked issues updated on the side of the application and the issue tracker (if any connected). | FR-15 | OK |
| AT-10 | AT-11 | View test runs. | Test runs shown. | FR-11 | OK |
| AT-10 | AT-12 | View reports. | Test reports shown. | FR-11 | OK |
| AT-10 | AT-13 | View generated issues, if any. | Generated issues shown. | FR-09 | OK |
| AT-10 | AT-14 | Add issue tracker information to the project. Link a bug to an issue tracker bug. | Project information updated. The bug is linked to the issue tracker bug. | FR-10 | OK |
| AT-5 | AT-15 | View and manage the test library (test plan, test, case, test module, test step definitions.) | Requested resource(s) shown. Requested action executed. | FR-06 | OK |
| AT-1 | AT-16 | Trigger a new test run. | The test run was triggered. | FR-12 | OK |

Table 7.1: Part I. of the acceptance testing table. The table describes possible basic scenarios/actions within TestBuDDy and connects them with the requirements (column *requirement coverage*).

The tables were also split by their importance whilst the requirement groups from chapter 3.3 were taken into consideration.

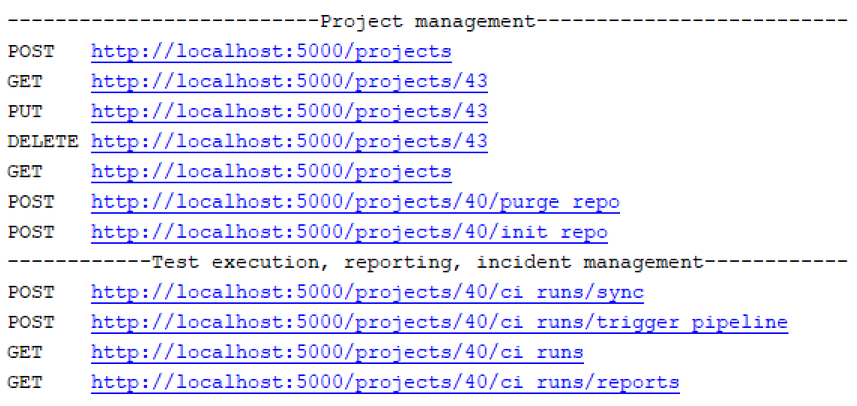
| **depen -dency** | **id** | **scenario** | **expected result** | **req. cover-age** | **state** |
|---|---|---|---|---|---|
| (AT-1) | AT-16 | Manage projects (create, modify, delete). | Requested action executed. |  | OK |
|  | AT-17 | Manage users (create, modify, delete). | Requested action executed. |  | OK |
| AT-2 | AT-18 | Manage user roles (create, modify, delete). | Requested action executed. |  | OK |
| AT-17 | AT-19 | Assign a role to a user. | The role assigned to the user. |  | OK |

| **depen -dency** | **id** | **scenario** | **expected result** | **req. cover-age** | **state** |
|---|---|---|---|---|---|
| AT-3 AT-17 | AT-20 | Assign the test case to a user. | The test case assigned to the user. | | OK |
| | AT-21 | View assigned roles and test cases for the user. | Requested resource shown. | FR-01 | OK |
| AT-1 | AT-22 | Manage requirements (create, modify, delete). | Requested action executed. | FR-16 | OK |
| AT-3 AT-22 | AT-23 | Assign a requirement to the test case. | The test case has the requirement assigned. | FR-17 | OK |
| AT-1 | AT-24 | Manage tags (create, modify, delete). | Requested action executed. | FR-16 | OK |
| AT-1 | | Tag a requirement. | The tag assigned to the requirement. | FR-16 | OK |

Table 7.2: Part II. of the acceptance testing table.

## 7.2 Automated Test Suite

TestBuddy is equipped with an automated set of unit tests for each endpoint and all its supported HTTP methods. The automated test suite can be found in file `tests.py` (in `backend` image) and is executed once the TestBuDDy application has started as part its docker container `unittest`. Each endpoint is tested from a *happy day* scenario perspective in order to cover the whole range of possible endpoint-HTTP method combinations. The endpoints mutually depend on each other, thus for example in order to access test case, project, test plan and test module have to be correctly initialised and connected. A positive response with status code 200 is always expected. An excerpt of an output of the automated test suite is in figure 7.1. An example of the full output of the automated test suite is in appendix E. The test suite is consists of endpoint groups (based on groups from chapter 3.3) and final cleanup of the database and the tested repository[1].



```
--------------------------Project management--------------------------
POST   http://localhost:5000/projects
GET    http://localhost:5000/projects/43
PUT    http://localhost:5000/projects/43
DELETE http://localhost:5000/projects/43
GET    http://localhost:5000/projects
POST   http://localhost:5000/projects/40/purge_repo
POST   http://localhost:5000/projects/40/init_repo
------------Test execution, reporting, incident management------------
POST   http://localhost:5000/projects/40/ci_runs/sync
POST   http://localhost:5000/projects/40/ci_runs/trigger_pipeline
GET    http://localhost:5000/projects/40/ci_runs
GET    http://localhost:5000/projects/40/ci_runs/reports
```

Figure 7.1: Example of the output of the automated test suite.

[1]This repository is also used as part of the initialisation for the TestBuDDy demonstration, please see section B.3 from appendix for further details.

## 7.3 Usability Testing

TestBuddy was tested by 5 subjects. The goal of the usability testing was to evaluate the application user flow. Each subject was given 1 complex scenario that represented the main focus of the application: BDD declarations generation and test run management. Each subject had a technical background and was presented with TestBuDDy general information, given Swagger doc, an example of a JBehave scenario (similar to the one from figure 6.3) and had to perform following tasks:

1. Initialise your data: `<root_url>/init-data`
2. Initialise your chosen project repository[2]: `<root_url>/projects/<proj_id>/init_repo`
3. View test plans.
4. Modify test module information.
5. Create a test case.
6. Add test steps for your created test case (see the provided scenario for example).
7. Synchronise your test run library.
8. View reports.
9. View bugs.

Table 7.3 describes the usability testing process. Results of usability testing are:

- Swagger doc needs to be fully documented into further details.
- Synchronisation of test run library, reports and issues has to be more visible within the thesis and thesis should clearly document the difference between the simple listing of current saved test runs and the synchronised test runs.
- It is recommended to create a frontend design for future expansions.

These results were incorporated during the thesis documentation period. In case of future application expansion of frontend, the next targeted user group are users with no technical background.

[2]User was expected to first view list of projects.

| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **observation** | **feedback** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **U1** | X | X | X | X | X | X | X | X | X | | *I knew what you were working on so I already expected the user flow. Worked OK.* |
| **U2** | X | X | X | X | - X | - - X | X | X | X | Needed help to create test case. The provided scenario was not descriptive enough. | *Swagger doc was OK, but the expected endpoints inputs should have mandatory fields. Full endpoint paths were quite verbose. Provided scenario was a bit puzzling, application only returns some parsing error.* |
| **U3** | X | X | X | X | X | X | - X | X | X | First wanted to see the test runs before their synchronisation. | *The synchronization is done only if I request it, maybe some synchronous sync could help. I liked working with test library. Could use frontend.* |
| **U4** | X | X | X | - | - | X | - X | X | X | Used other already created test case. First wanted to see the test runs before their synchronisation. | *I'm not really into web app design, although I really liked the connection with repository and Gitlab integration.* |
| **U5** | X | X | X | X | X | X | X | X | - X | Synchronized test runs and reports, but not issues at first, had to go back to sync. | *The sync endpoint Swagger doc was not sufficient documentation. Overall JSON syntax is helpful, verbose.* |

Table 7.3: Usability testing. Scenario tasks are marked by numbers, users as *U*. *Observation* represents TestBuDDy creator's point of view during user's work on tasks, *feedback* represents user's shortened feedback. *X* represents success, - represents a failed try.

# Chapter 8

# Conclusion

The aim of this thesis was to specify, design and implement a new BDD orientated test management tool based on the analysis of existing proprietary and open source competitors.

First, a theoretical part discussed required testing terminology with the emphasis on the BDD process. Also the existing test management tools were analysed, categorised and compared alongside integrated tools for issue tracking and BDD approach. The analysed findings provided information to help to determine the final list of requirements for the application where the requirements were set and categorised with the help of Use case diagrams. Additionally, the technologies and languages that were used in implementation were analysed, compared and selected whilst taking into consideration the application architecture. The application design described the application from various perspectives: from the general architecture and REST API, through OOP design, up to the main focus of the application: the test library management and the test run reporting. The test library is discussed including its connection to the source code of tests on the SUT (*Software under test*) remote repository. Next, the chosen integrations for the BDD framework *JBehave* and the CI/CD tool *Gitlab CI* were discussed and possible problematic scenarios that could occur during implementation were addressed. The following chapter digressed into application implementation details and connected the test library with the user flow and devised procedures and resolutions for emerged test library issues. Another important part of implementation was data synchronisation with the CI/CD tool. Based on acquired data, the reports are created and issues are analysed and generated. Finally, the testing of the resulting application was described and its automated test suite documented.

The final output of this thesis is the **TestBuDDy** application which is implemented as a web service and offers a way to ease up workload of a testing team. Project managers or other team members with small technical backgrounds can use TestBuDDy to input business requested BDD scenarios to their software used to test SUT. TestBuddy creates all the needed source code for the BDD framework (that testing team had chosen) and updates the SUT repository to correspond with its current test library. The CI/CD tool that manages the repository then runs the new version of the SUT test suite. TestBuDDy generates BDD declarations for each test step definition and also their failing BDD definitions as part of the test driven development approach. Test engineers with a technical background are subsequently expected to input the BDD definitions and update the project remote repository. TestBuDDy is able to acquire repository test run data from the CI/CD tool. During this synchronisation, the application creates test run reports, analyses them and automatically generates and groups found issues. These issues can be further synchronised with an integrated issue tracker, JIRA at current.

Although the application offers said features, the first major possible future expansion could be to add a frontend GUI, that was already outlined and shown in the design section based on findings from usability testing. A frontend GUI would hide verbose REST API endpoint design and the system could control more the user actions within the application. Another possible expansion is to build upon the current basic TestBuDDy user and role management functionality - also implement authentication and access control within the GUI. The second larger expansion would be to fully support application scalability. At present, the endpoints use and modify the core functionality, but it would be beneficial to further divide the core functionality so that the endpoints would only use this core functionality (without any modifications on the endpoint layer). A third major expansion could be related to supported integrations - broaden the support for other BDD frameworks, CI/CD tools and issue trackers, so that TestBuDDy could measure up to commercial software. A fourth bigger expansion could be to implement selective test run management, where the user would actively choose which test case, test module, etc. should be executed as opposed to the current state where the whole suite is executed as a *batch*. Other possible expansions could also be to add support for more test plans in relation to the project and support of different *git branches*.

To summarise the **system features**: TestBuDDy provides means to manage the whole test library (test plans, test cases, test steps, test step definitions), test run library, its reports and issues. Test library management is successfully projected onto a BDD framework of a SUT project repository. The application is able to be synchronised with the CI/CD tool and also the issue tracker. The application also allows users to manage their projects, project requirements and provides support for user and role management in regard to possible expansions in the future. Based on the aforementioned system features, the resulting application is in concordance with the thesis assignment. TestBuDDy has the potential to help testers during their work flow, saving time and effort.

# Bibliography

[1] *Best 25 Test Management Tools in 2019.* [Online; visited 03.01.2019].
Retrieved from: https://www.guru99.com/top-20-test-management-tools.html/

[2] *Bugzilla vs JIRA vs Redmine System Properties Comparison.* [Online; visited 07.01.2019].
Retrieved from:
https://project-management.zone/system/bugzilla,jira,redmine/

[3] *DB–Engines Ranking.* [Online; visited 03.01.2019].
Retrieved from: https://db-engines.com/en/ranking/

[4] *Historical trends in the usage of client-side programming languages for websites.* [Online; visited 03.01.2019].
Retrieved from: https://w3techs.com/technologies/history_overview/client_side_language/all

[5] *Issue Management Tools - Popularity Ranking. [Online; visited 07.01.2019].
Retrieved from: https://project-management.zone/ranking/category/issue/*

*[6] PHP Usage Statistics. [Online; visited 07.01.2019].
Retrieved from: https://trends.builtwith.com/framework/PHP/*

*[7] AMMAN, P.; OFFUTT, J.: Introduction to software testing. Cambridge University Press. 2008. ISBN 978-0-521-88038-1.*

*[8] CONDO, C.; LECLAIR, A.: The Forrester Wave: Continuous Integration Tools, Q3 2017. 2017.*

*[9] DASSO, A.; FUNES, A.:* Verification, Validation and Testing in Software Engineering. *IGI Global. 2007. ISBN 978-1-59140-851-2.*

*[10] DWYER, G.: Flask vs. Django: Why Flask Might Be Better. February 2017. [Online; visited 06.01.2019].
Retrieved from: https://www.codementor.io/garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7mdf8v/*

*[11] FREEMAN, E.; ROBSON, E.; BATES, B.; et al.:* Head first design patterns. *O'Reilly Media, Inc.. 2004.*

*[12] KANER, C.; FALK, J.; QUOC NGUYEN, H.:* Testing computer software. *Van Nostrand Reinhold. second edition. 1993. ISBN 978-0-442-01361-5.*

[13] *MEYER, M.: Continuous Integration and Its Tools.* IEEE Software. *vol. 31, no. 3. May 2014: pp. 14–16. ISSN 0740-7459. doi:10.1109/MS.2014.58.*

[14] *PATTON, R.:* Software testing. *Indianapolis: Sams Publishing. second edition. 2006. ISBN 0-672-32798-8.*

[15] *ROBINSON, D.: The Incredible Growth of Python. September 2017. [Online; visited 07.01.2019].*
*Retrieved from:*
https://stackoverflow.blog/2017/09/06/incredible-growth-python/

[16] *SOLIS, C.; WANG, X.: A Study of the Characteristics of Behaviour Driven Development. In* 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. *Aug 2011. ISSN 1089-6503. pp. 383–387. doi:10.1109/SEAA.2011.76.*

[17] *SOMMERVILLE, I.:* Software engineering 9th edition. *Edinburgh: Addison-Wesley. 9 edition. 2011. ISBN 978-0-13-703515-1.*

[18] *SORAPAK, P.: The Comparative Study of Collaborative Learning and SDLC Model to develop IT Group Projects.* TEM Journal. *vol. 6, no. 4. 2017: pp. 800–809. ISSN 2217-8309. doi:10.18421/TEM64-20.*
*Retrieved from:* http://search.proquest.com/docview/2155128342/

[19] *SPILLNER, A.:* Software testing foundations : a study guide for the certified tester exam: foundation level, ISTQB compliant. *Santa Barbara, CA: Rocky Nook. fourth edition. 2014. ISBN 978-1-937-53842-2.*

[20] *WYNNE, M.; HELLESØY, A.:* The Cucumber Book: Behaviour-Driven Development for Testers and Developers. *Santa Barbara, CA: Pragmatic Programmers, LLC. second edition. 2012. ISBN 978-1-934356-80-7.*

# Appendix A

# Content of Enclosed DVD

Enclosed DVD contains following files[1]:

- `doc` folder - to store documentation
- **src** folder - to store project files:

```
/src
 ├─ backend............................ project sources (includes tests)
 │   ├─ ci_communication.................................. CI/CD tool connection
 │   ├─ doc............................... Swagger doc, documentation
 │   ├─ endpoints.......................................... all endpoint requests
 │   ├─ issue_tracking.................................. issue tracker connection
 │   └─ processing................................ BDD framework connection
 │       └─ jbehave_src............................................ JBehave files
 ├─ database................................ persistent database folder
 └─ dbs..................................... database initialising script
     └─ init_dbs.sql
```

[1]The detailed file structure is also described and shown in section 6.1.

# Appendix B

# Installation and Manual

TestBuddy sources are available in enclosed CD, folder `src/`.

## B.1 Prerequisites

Application is multiplatform, but it is recommended to use Windows type of an operation system. Users need to have *Docker* and a modern web browser installed on their local machine. In case of Linux system, *docker* and *docker-compose* are necessary dependencies. Please copy the `src/` folder from the enclosed DVD.

## B.2 Deployment

A user is expected to use following commands to correctly start TestBuDDy (its containers for the backend, the database and tests) from `src/`.

- `docker-compose build` - first, the docker containers have to be built.
- `docker-compose up` - to start TestBuDDy; in case when the user wants to not only start the application, but also execute its unit tests (for endpoints)[1].
- `docker-compose down` - to stop the application and destroy the containers.
- `docker-compose up backend` - to only start TestBuDDy application (with no changes on the database).
- `docker-compose up --build` - to start TestBuDDy, run tests and build the application containers in one command

User can view all REST API endpoints in *Swagger API documentation* on `root_url` of the project, which is set to `127.0.0.1:5000`. An example of the Swagger documentation is also in appendix D. An example of available endpoints are also visible form the output of the automated test suite in appendix E. Swagger allows user to test every endpoint and defines expected results of a request call. Is is also possible to use API development tool *Postman*[2], alternatively *cUrl* CLI tool[3].

[1] The test suite duration can take up around 2 minutes. The test suite also cleans the application database.
[2] Available at: `https://www.getpostman.com/`.
[3] Available at: `https://curl.haxx.se/`.

## B.3 Demo initialisation

### Test Library

For demonstrative purposes, there is an **initialising endpoint** (POST type of a request) provided:

```
127.0.0.1:5000/init-data
```

2 active public project repositories are already set up for the user when working with their test library:

- https://pajda.fit.vutbr.cz/testos/testbuddy-sut
- https://pajda.fit.vutbr.cz/xblozo00/testing_repo

Upon the creation of both projects, TestBuDDy uses the Gitlab CI generated token, that will be available for next few months. It is also recommended to first initialise your projects' repository by the following endpoint of a POST type:

```
127.0.0.1:5000/projects/<proj_id>/init_repo
```

The project has its test library already created until the test case level (in the database layer), so a user can view current test cases and add a new scenario of their choice, or,if preferred, create their own test cases and so on.

### Test Run Management and Gitlab CI

The repositories were already used in the past, so when working with its test run library, there are also historic pipeline records and reports processed and shown. These records are shown when synchronising with Gitlab CI by calling an endpoint of a POST type:

```
127.0.0.1:5000/projects/<proj_id>/ci_runs/sync
```

### Requirement Management

Requirements and tags were also created, so user is free to manage them, connect them with existing test cases etc.

### Incident Management and JIRA Integration

For the purpose of TestBuDDy demonstration of incident management, JIRA page was created: https://tesbuddy.atlassian.net/. User can log in using these credentials:

- username: *testbuddyBUT@gmail.com*
- password: *gherkinisawesome*

# Appendix C

# Database Model

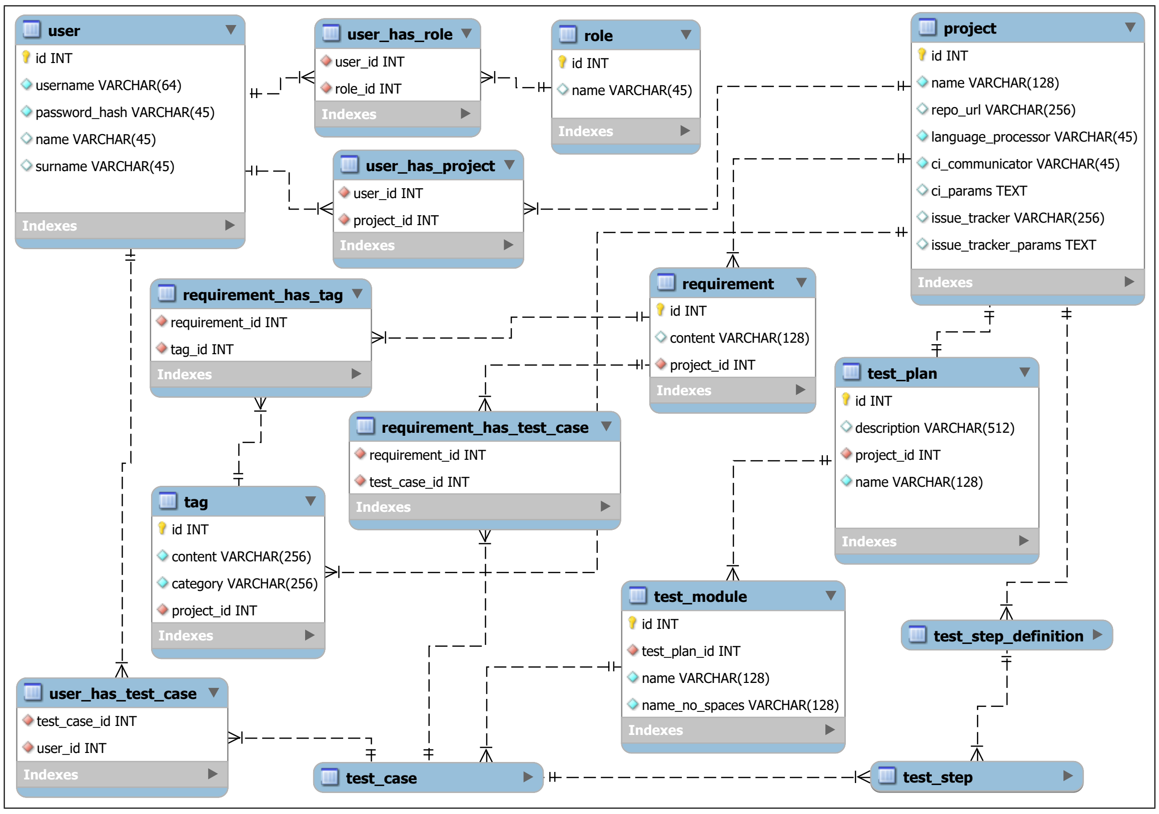


Figure C.1: Application database section I.: User, project and requirement management groups + connection to model.

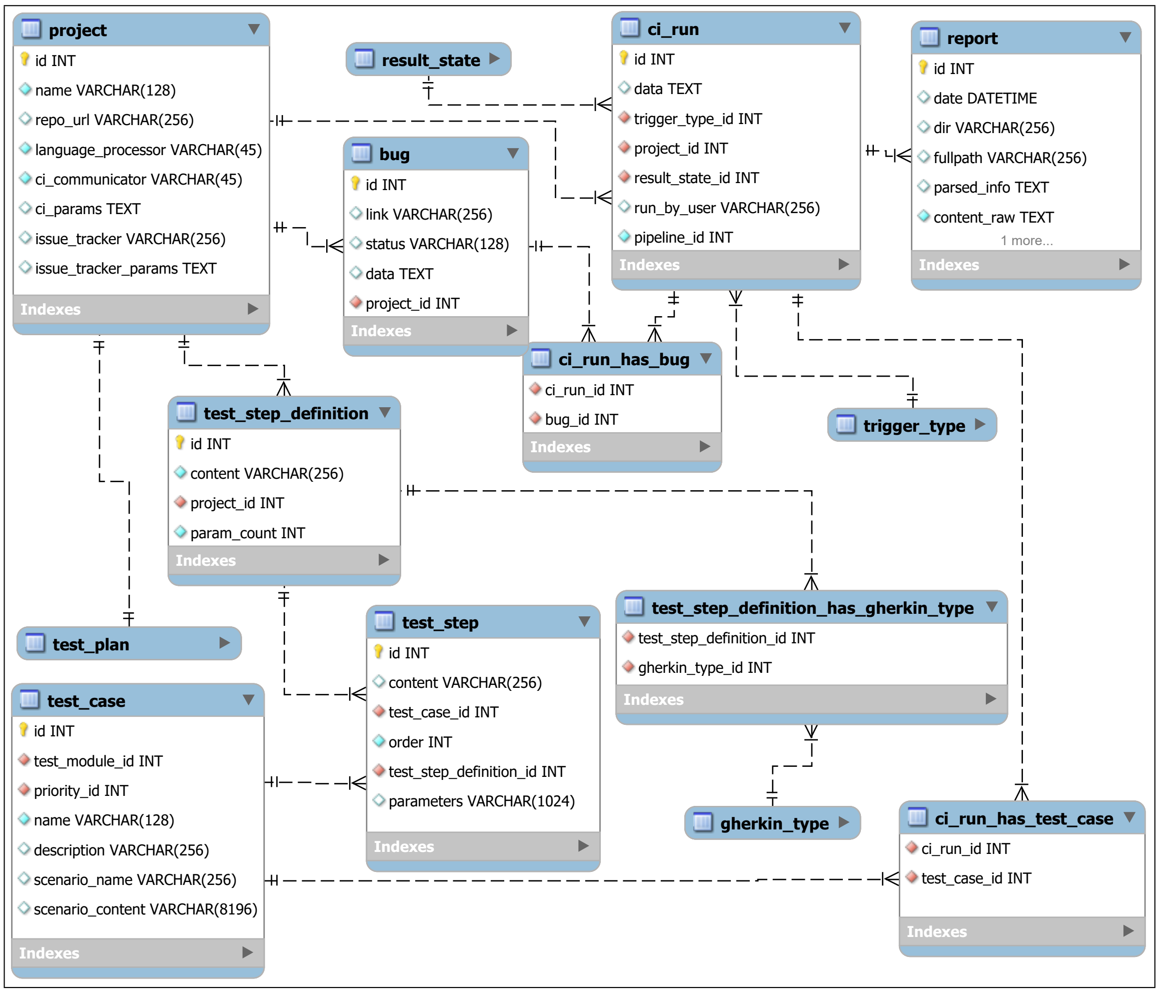


Figure C.2: Application database section II.: Test specification, test execution, report and incident management groups + connection to model.

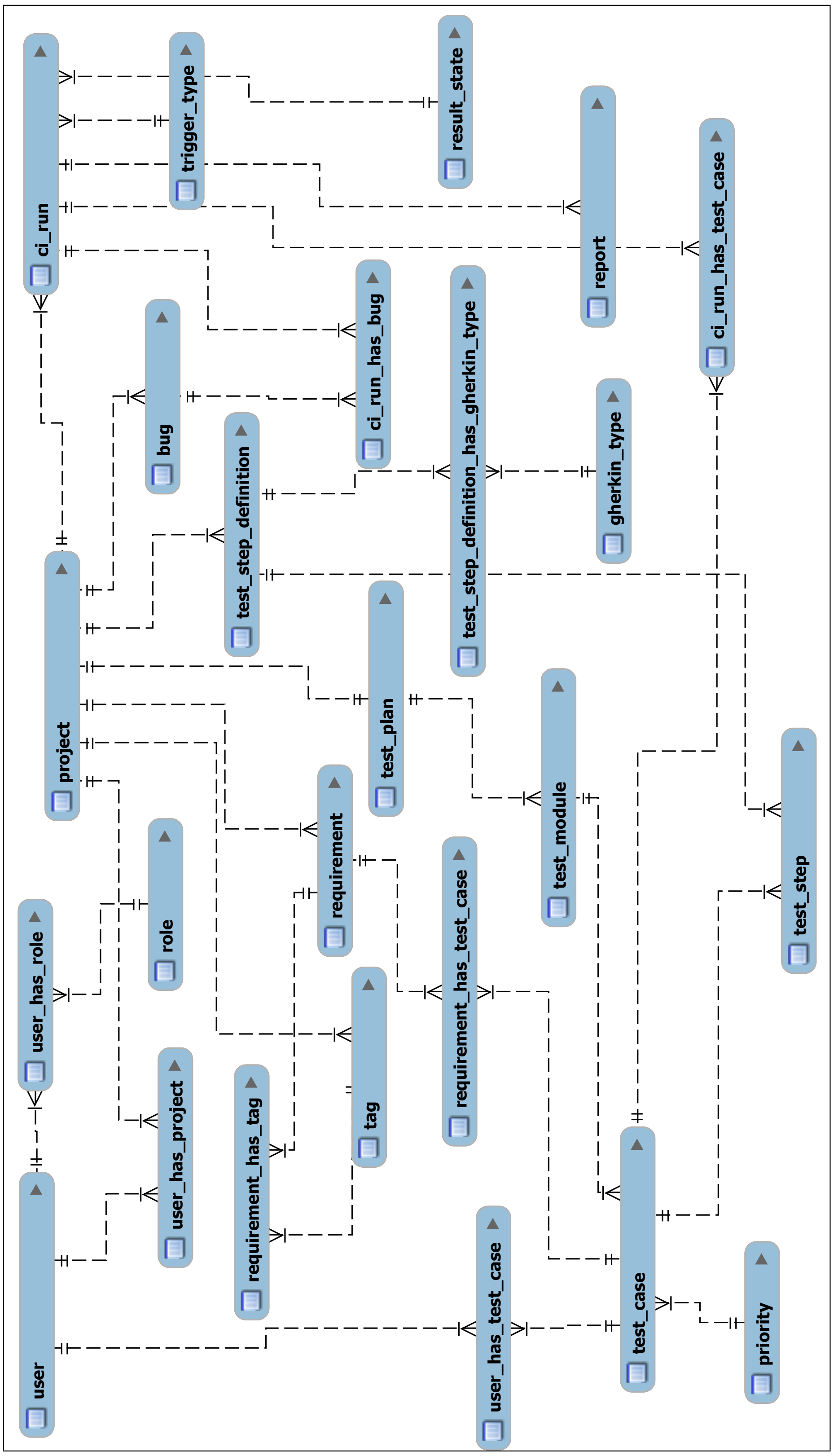
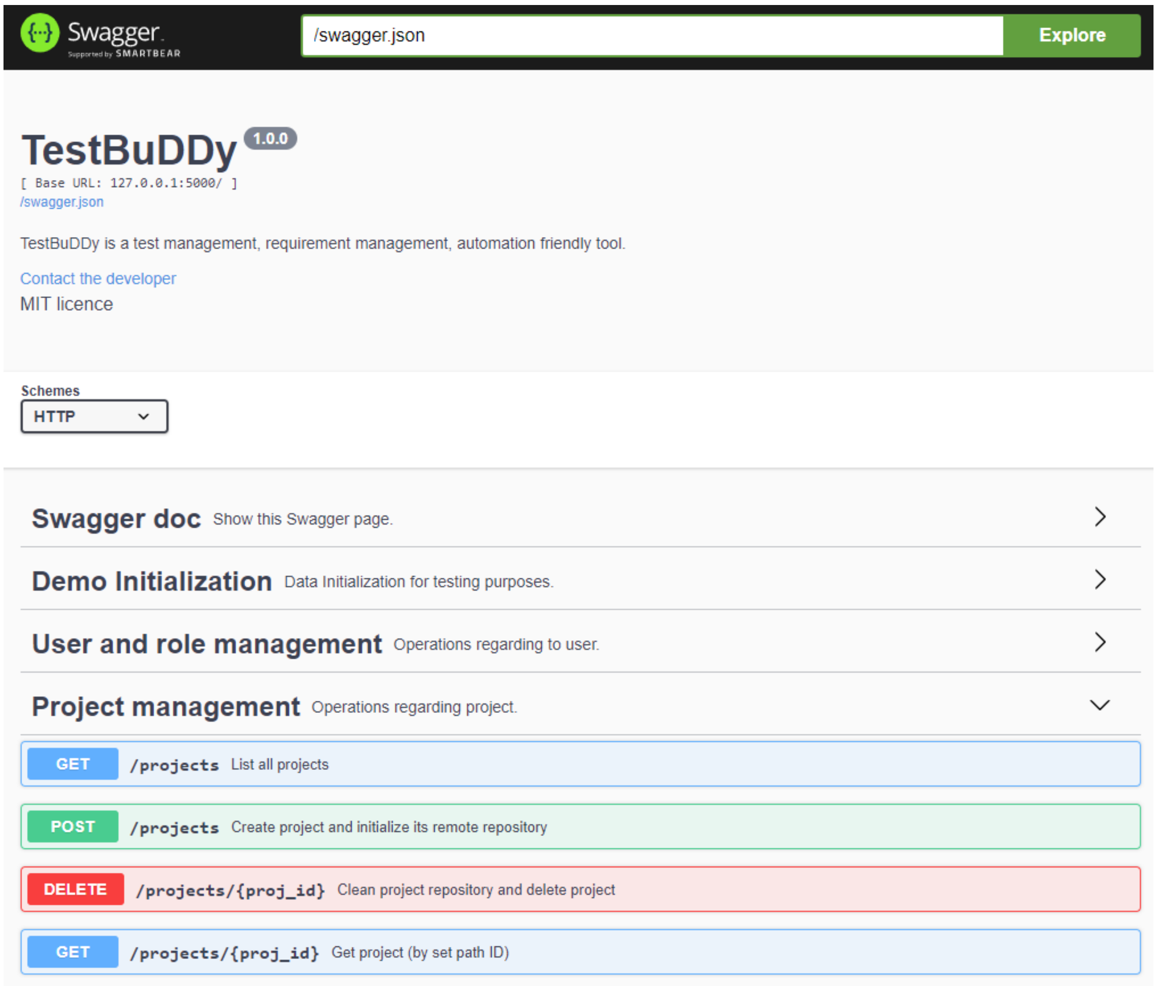


Figure C.3: ER diagram of overall database schema.

# Appendix D

# Swagger Documentation Example



/swagger.json

TestBuDDy 1.0.0

[ Base URL: 127.0.0.1:5000/ ]

/swagger.json

TestBuDDy is a test management, requirement management, automation friendly tool.

Contact the developer

MIT licence

Schemes: HTTP

Swagger doc — Show this Swagger page.

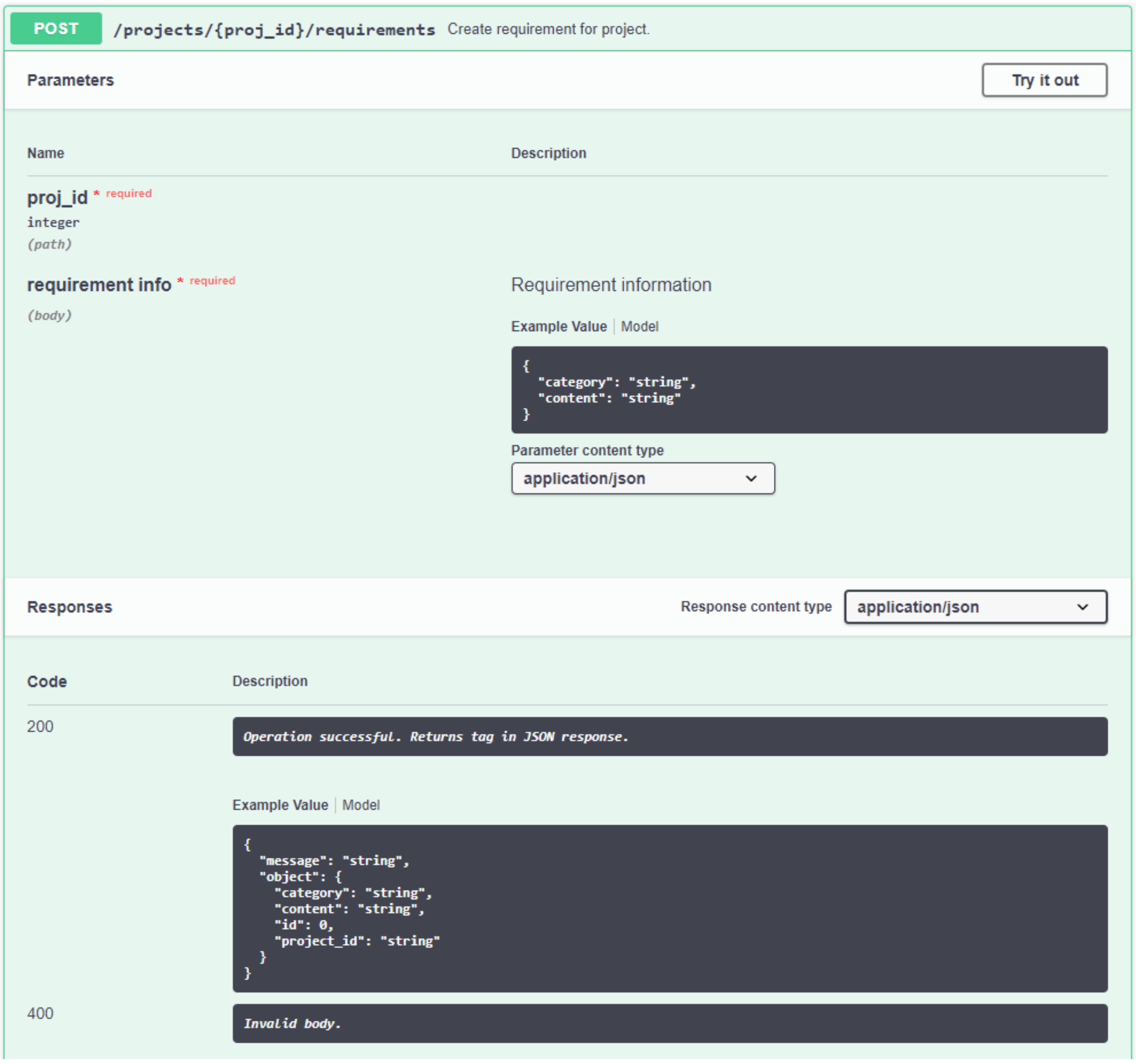
Demo Initialization — Data Initialization for testing purposes.

User and role management — Operations regarding to user.

Project management — Operations regarding project.

```
GET     /projects              List all projects
POST    /projects              Create project and initialize its remote repository
DELETE  /projects/{proj_id}    Clean project repository and delete project
GET     /projects/{proj_id}    Get project (by set path ID)
```

Figure D.1: Example of incorporated Swagger doc for all endpoints in root url of TestBuDDy.



`/projects/{proj_id}/requirements` Create requirement for project.

Parameters

| Name | Description |
|---|---|
| proj_id * required<br>integer<br>*(path)* | |
| requirement info * required<br>*(body)* | Requirement information |

```
{
  "category": "string",
  "content": "string"
}
```

Parameter content type: application/json

Responses

Response content type: application/json

| Code | Description |
|---|---|
| 200 | *Operation successful. Returns tag in JSON response.* |
| 400 | *Invalid body.* |

```
{
  "message": "string",
  "object": {
    "category": "string",
    "content": "string",
    "id": 0,
    "project_id": "string"
  }
}
```

Figure D.2: Detail of Swagger endpoint - a creation of a requirement. A user is expected to fill in the path parameters and the JSON body of this POST type request. Request has expected responses and their codes already predefined.

# Appendix E

# Output of The Automated Test Suite

Example of the full output of the automated test suite - unit tests of all available endpoint-HTTP methods within TestBuDDy.

```
--------------------------TESTBUDDY---TESTS--------------------------
--------------------------Demo endpoints------------------------------
GET    http://localhost:5000
POST   http://localhost:5000/clean-data
POST   http://localhost:5000/init-ci
POST   http://localhost:5000/init-projects
POST   http://localhost:5000/init-requirements
POST   http://localhost:5000/init-testlibrary
POST   http://localhost:5000/clean-data
POST   http://localhost:5000/init-data
--------------------------Project management--------------------------
POST   http://localhost:5000/projects
GET    http://localhost:5000/projects/173
PUT    http://localhost:5000/projects/173
DELETE http://localhost:5000/projects/173
POST   http://localhost:5000/projects
GET    http://localhost:5000/projects
POST   http://localhost:5000/projects/170/purge_repo
POST   http://localhost:5000/projects/170/init_repo
------------Test execution, reporting, incident management------------
POST   http://localhost:5000/projects/170/ci_runs/sync
POST   http://localhost:5000/projects/170/ci_runs/trigger_pipeline
GET    http://localhost:5000/projects/170/ci_runs
GET    http://localhost:5000/projects/170/ci_runs/reports
GET    http://localhost:5000/projects/170/ci_runs/2003
GET    http://localhost:5000/projects/170/ci_runs/2003/reports
GET    http://localhost:5000/projects/170/bugs
GET    http://localhost:5000/projects/170/bugs
GET    http://localhost:5000/projects/170/bugs
----------------------------Test library------------------------------
GET    http://localhost:5000/projects/plans
GET    http://localhost:5000/projects/170/plans
POST   http://localhost:5000/projects/174/plans
PUT    http://localhost:5000/projects/170/plans/117
POST   http://localhost:5000/projects/170/plans/117/mods
PUT    http://localhost:5000/projects/170/plans/117/mods/638
DELETE http://localhost:5000/projects/170/plans/117/mods/638
GET    http://localhost:5000/projects/170/plans/117
GET    http://localhost:5000/projects/170/plans/117/mods/636/cases
POST   http://localhost:5000/projects/170/plans/117/mods/636/cases
POST   http://localhost:5000/projects/170/plans/117/mods/636/cases/844
PUT    http://localhost:5000/projects/170/plans/117/mods/636/cases/844
GET    http://localhost:5000/projects/170/stepdefinitions
GET    http://localhost:5000/projects/170/stepdefinitions/25/steps
DELETE http://localhost:5000/projects/170/plans/117/mods/636/cases/844
------------------------Requirement management------------------------
GET    http://localhost:5000/projects/170/tags
POST   http://localhost:5000/projects/170/tags
GET    http://localhost:5000/projects/170/tags/488
PUT    http://localhost:5000/projects/170/tags/488
GET    http://localhost:5000/projects/170/requirements
POST   http://localhost:5000/projects/170/requirements
GET    http://localhost:5000/projects/170/requirements/1515
PUT    http://localhost:5000/projects/170/requirements/1515
POST   http://localhost:5000/projects/170/requirements/1515/assign-tag
DELETE http://localhost:5000/projects/170/requirements/1515/assign-tag
POST   http://localhost:5000/projects/170/plans/117/mods/636/cases/845/requirements/ass⌋
 ↪  ign-requirement
GET    http://localhost:5000/projects/170/plans/117/mods/636/cases/845/requirements
DELETE http://localhost:5000/projects/170/plans/117/mods/636/cases/845/requirements/ass⌋
 ↪  ign-requirement
DELETE http://localhost:5000/projects/170/tags/488
DELETE http://localhost:5000/projects/170/requirements/1515
----------------------User and role management------------------------
GET    http://localhost:5000/users/roles
POST   http://localhost:5000/users/roles
GET    http://localhost:5000/users/roles/78
PUT    http://localhost:5000/users/roles/78
GET    http://localhost:5000/users
POST   http://localhost:5000/users
GET    http://localhost:5000/users/95
PUT    http://localhost:5000/users/95
POST   http://localhost:5000/users/95/assign-project
DELETE http://localhost:5000/users/95/assign-project
POST   http://localhost:5000/users/95/assign-role
DELETE http://localhost:5000/users/95/assign-role
POST   http://localhost:5000/users/95/assign-testcase
DELETE http://localhost:5000/users/95/assign-testcase
DELETE http://localhost:5000/users/roles/78
DELETE http://localhost:5000/users/95
--------------------------Final cleanup-------------------------------
POST   http://localhost:5000/projects/170/purge_repo
POST   http://localhost:5000/clean-data
--------------------------TESTBUDDY---TESTS--DONE---------------------
```