Multi-cloud Execution-ware for Large-scale Optimised Data-Intensive Computing

H2020-ICT-2016-2017
Leadership in Enabling and Industrial Technologies; Information and Communication Technologies

Grant Agreement No.:
731664

Duration:
1 December 2016 -
30 November 2019

www.melodic.cloud

Deliverable reference:
5.10

Date:
31 May 2017

Responsible partner:
7bulls

Editor(s):
Małgorzata Jakubczyk

Author(s)
Małgorzata Jakubczyk,
Marcin Prusiński

Approved by:
Ernst Gunnar Gran

ISBN number:
N/A

Document URL:
http://www.melodic.cloud/deliverables/D5.10 Quality Assurance Guide.pdf

Title:

Quality Assurance Guide

Abstract:

This document presents the most important processes and tasks related to Quality Assurance of software developed in Melodic. In particular, the document should be used as a manual for performing all activities within testing processes. Also it should be used in conjunction with the 'D5.06 Test Strategy and Environment' deliverable, which contains the strategy of testing, acceptance criteria, test related products and so on. The deliverable's intended audiences is as follows: It could be used by all participants in the project. Especially it shall be use by development teams (bugs handling process), test teams (bugs, test cases processes), architects (bugs, test cases processes), use case application users (test reporting, bugs handling) and management of the project (status of the testing of release). Altogether, the document is a complete and comprehensive manual for the most important software quality assurance tasks in the Melodic project.

This project has received funding from
the European Union's Horizon 2020 research
and innovation programme under grant agreement No 731664

Document	
Period Covered	M1-6
Deliverable No.	D5.10
Deliverable Title	Quality Assurance Guide
Editor(s)	Małgorzata Jakubczyk
Author(s)	Małgorzata Jakubczyk, Marcin Prusiński
Reviewer(s)	Kyriakos Kritikos, Daniel Seybold
Work Package No.	5
Work Package Title	Integration and Security
Lead Beneficiary	7bulls
Distribution	PU
Version	1.0
Draft/Final	Final
Total No. of Pages	41

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Table of Contents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

List of Figures

1 Introduction

The Quality Assurance Guide has been developed by the 7bulls team to describe the software quality assurance process in the Melodic project. The document has been prepared under task 5.4 of Quality Assurance. In the Melodic project, we have the following main goals:

- provide a high quality platform that allows to run data-intensive applications in a geographically distributed multi-cloud infrastructure
- ensure the security and privacy of data
- enable transparent deployment of applications on multi-cloud infrastructure
- minimize the risk of failure in a live environment

Detailed acceptance criteria are to be found in "D5.06 Test Strategy and Environment".

1.1 The scope

The Quality Assurance Guide contains the following information which has been structured in subsequent chapters:

- Software deployment environments – information about environments which will be used in the project
- Bugs management – information about process bug reporting, bug workflow, bug elements, priorities and labels
- Test Case Management – information about the process of test case creation, the test case workflow, elements, types and priorities
- Testing tools – information of tools which will be used in the project
- Delivery management – description of delivery management process
- Meetings – information of testing meetings in the Melodic project.

The deliverable ends with a short summary of its content in the final chapter.

1.2 Audience

This document is intended for those involved in the Melodic project. This includes project coordinators, architects, developers and the test team. The roles of an Architect, Developer, and Tester is described in detail in 'Description of Action' attached to the Grant Agreement.

1.3 Structure of the document

The rest of the document is organised as follows. Chapter 2 contains descriptions of all environments that will be using in the Melodic project, and describes which tests will be performed on which environment and by whom. In Chapter 3 we have information about the bug's management process and detailed information on how to create a bug, from what elements it is composed, and what priorities and labels it can have. Chapter 4 contains detailed information about test cases – similarly as for bug – including the process of creating test cases, and information about test elements, priorities and types. Chapter 5 provides information about the tools which will be used for the various types of tests, including unit, integration, load, performance and acceptance tests. Chapter 6 presents the process of release management, responsibility for the deployments and information of acceptability criteria. Chapter 7 provides information about meetings organized during the lifetime of the Melodic project. Finally, Chapter 8 concludes this deliverable.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

2 Software deployment environments

This chapter covers a short description of the testing environments and their scope, as well as some particular restrictions and guidelines that apply to them. There are three stages of testing, as can be seen in the following figure, which are directly related to the respective testing environments:

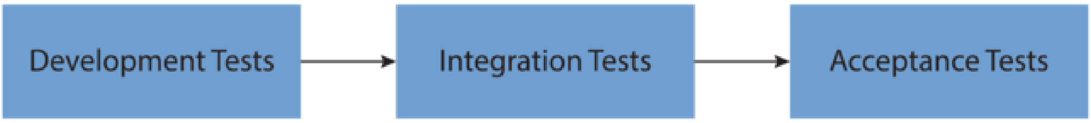


Figure 1 – List of environments

In development test environments, the tests will be performed only by developers. In integration test environments, the tests will be executed by developers and testers, and in acceptance test environments, the tests will be executed by testers.

2.1 Audience

This chapter is intended for those involved in the development:

- mandatory for architects and developers
- recommended for the test team
- optional for all

2.2 Development Tests environments

The purpose of Development Tests environments is to ensure that the code produced by a developer is executable before committing changes to the component's master branch repository. This kind of environment is suitable for performing Unit Tests. There should be at least one development test environment for each developer involved in the project. Each such environment should be created and maintained by the developer himself and software builds should be performed manually on demand. A developer's test environment should contain only Melodic modules needed for a certain development (as there is no need to maintain the whole Melodic platform).

2.3 Integration Tests environment

The purpose of an Integration Tests environment is to allow Melodic developers to integrate the components that they develop with stable releases of the rest of the Melodic components. There will be one integration environment created and maintained solely by 7Bulls including all of the project's application components. It will utilize Continuous Integration (CI)[1] practice with the use of the Jenkins[2] automation server. Software builds will be performed automatically from the master branch after each commit and an additional build will be executed on a nightly basis. This type of environment will be used to perform Functional Tests.

2.4 Acceptance Tests environment

An Acceptance Tests environment will be utilized by the test team to perform tests over final versions of the Melodic platform. Therefore, all project's components will be installed and used. The Acceptance Tests environment will also be solely maintained by 7Bulls with the use of Jenkins CI. The main difference is that builds will be performed not from the master, but from specific release (or release candidate) branches, when needed. At the beginning, smoke tests will be executed in Acceptance Tests environment. *Smoke testing* consists of a subset of test cases from functional testing, executed at the beginning of the functional phase to ensure that the system is stable. In the next step, with the use of this test environment all of the following types of tests will be performed: Load, Security and Acceptance. More information about level of testing are to be found in 'D5.06 Test Strategy and Environment'.

[1] https://en.wikipedia.org/wiki/Continuous_integration

[2] https://jenkins.io/

3 Bugs management

This chapter contains detailed information about the management and resolving of bugs in the Melodic project. In particular, it contains information about the process of bug reporting as well as the bug flow, elements, priorities and labels.

During the development and testing of the system, bugs can be found. A bug is a defect in a component or system that can cause the component or system to not perform its required function. A defect that will be encountered during runtime, can cause failure of the component or system. It constitutes a deviation of the component or system from its expected delivery, service or result. Bugs are reported in order to enable the system to subsequently work as planned and expected, when they are appropriately corrected. Bugs are corrected by developers and checked by testers. More information about bugs can be found in chapter 3.4 - "Bugs life cycle".

3.1 Audience

This chapter is intended for those involved in the development. It is

- mandatory for architects and developers, as developers will fix a bug, architects will explain any ambiguity about the bug, while both have to know the bug management workflow.
- recommended for all

3.2 Responsibility

Based on its current state – *Open, Blocked, In progress, To test, Closed* – a bug will have to be handled by a specific role. Such responsibilities along with their mapping to respective roles are provided below:

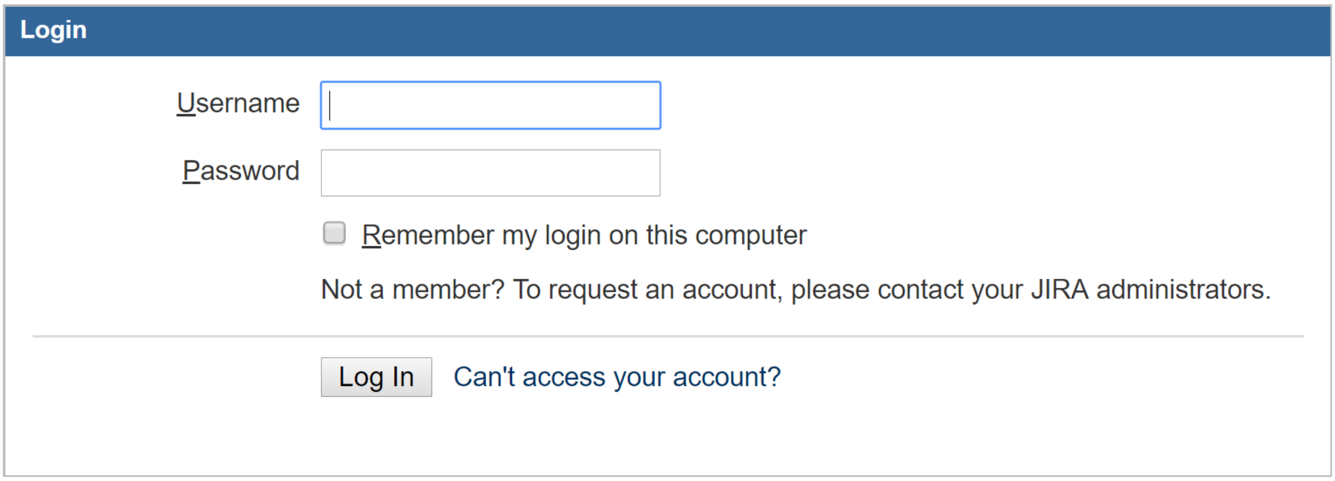
- Architects – handling of bugs with status *Blocked*
- Developers – handling of bugs with status *Open, Blocked, In progress*
- Test team – handling of bugs with status *To test*

Detailed information about the roles for each state of a bug is described in chapter 3.4 - "Bugs life cycle".

3.3 Bug Reporting Process

Bugs will be reported in the JIRA[3] system as an issue of type 'Bug'. Such bug issues will be mainly created by a Test team comprising a test leader and a set of testers. Bugs can also be created by other team members – developers, architects, leaders. Bugs will be created during development, integration, security, performance and acceptance tests. Please see below the process of how to create a bug in the JIRA system:

1. In a browser, open the page: https://jira.7bulls.eu
2. Enter correct values to fields 'Username' and 'Password' and press the 'Log In' button:



Login

Username

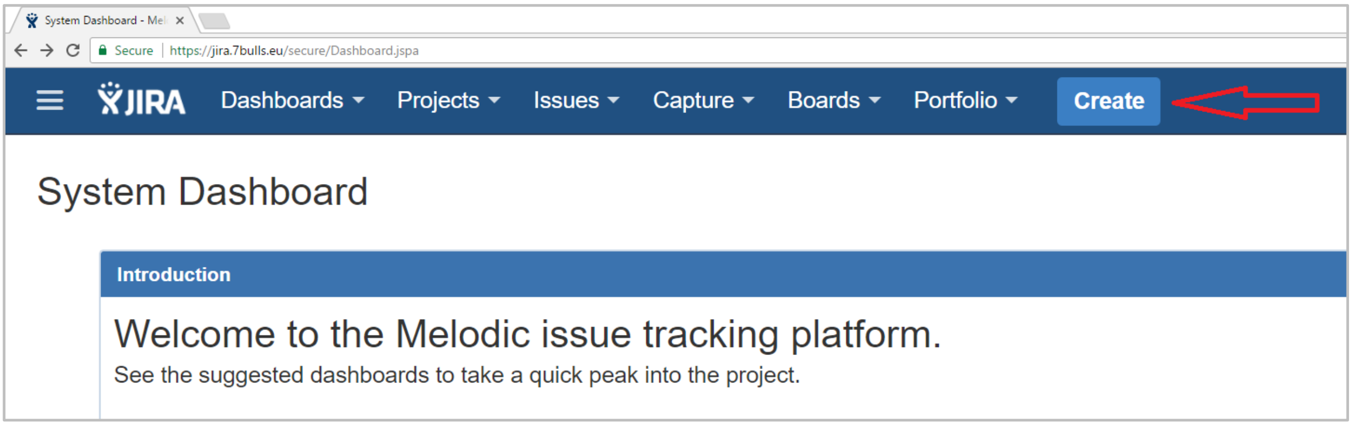
Password

□ Remember my login on this computer

Not a member? To request an account, please contact your JIRA administrators.

Figure 2 – JIRA Login Page

3. To create a new issue, press 'Create':



System Dashboard

Introduction

Welcome to the Melodic issue tracking platform.

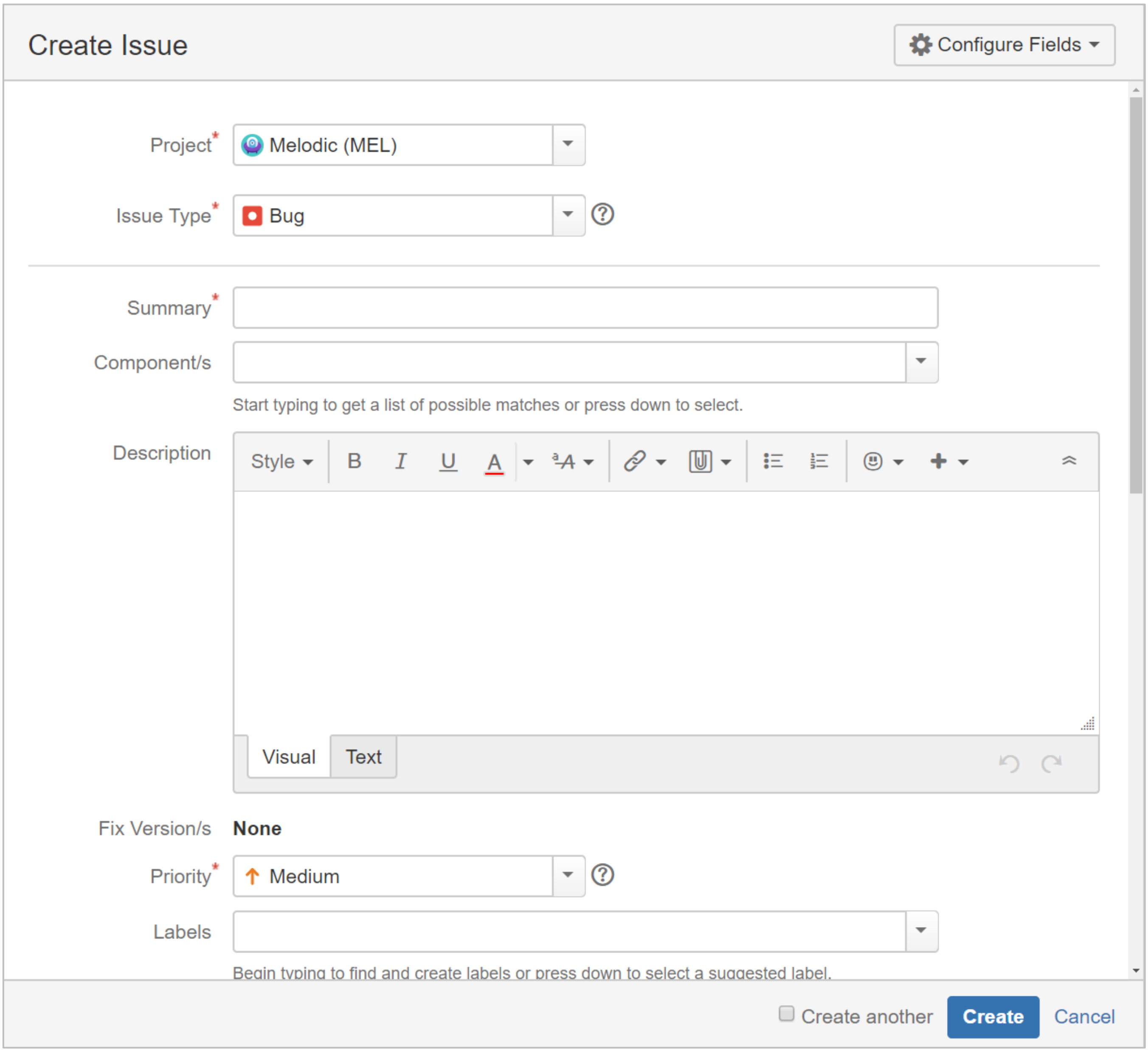
See the suggested dashboards to take a quick peak into the project.

Figure 3 – JIRA Home Page

[3] https://www.atlassian.com/software/jira

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

4. Fill in all needed fields:



Create Issue

Project* Melodic (MEL)

Issue Type* Bug

Summary*

Component/s

Start typing to get a list of possible matches or press down to select.

Description

Fix Version/s **None**

Priority* Medium

Labels

Begin typing to find and create labels or press down to select a suggested label.

Figure 4 – JIRA Create Issue page

5. Press the 'Create' button.

After following these steps, the new bug will be created. Information about bug required fields can be found in chapter 3.5 - "Bug's elements". The bug handling process is independent of regression. We can report the bug in regression tests as well as in normal tests, and the same handling is followed regardless of the status. Regression tests involve testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software as a result of the changes made. It is performed when the software or its environment is changed.

In case we desire to edit a created bug, the process below can be followed.

Identify the bug by either:

1. Entering the issue number in the search box and pressing the 'Enter' button:

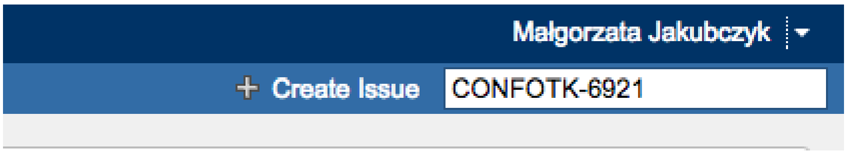


Figure 5 - JIRA Search Issue

or

2. Searching the issue via:
   a. pressing the arrow next to the 'issue' button
   b. choosing the option "Search for issues"

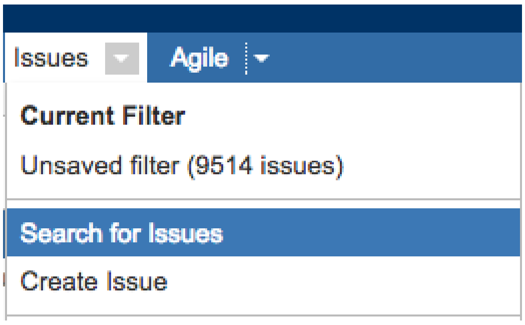
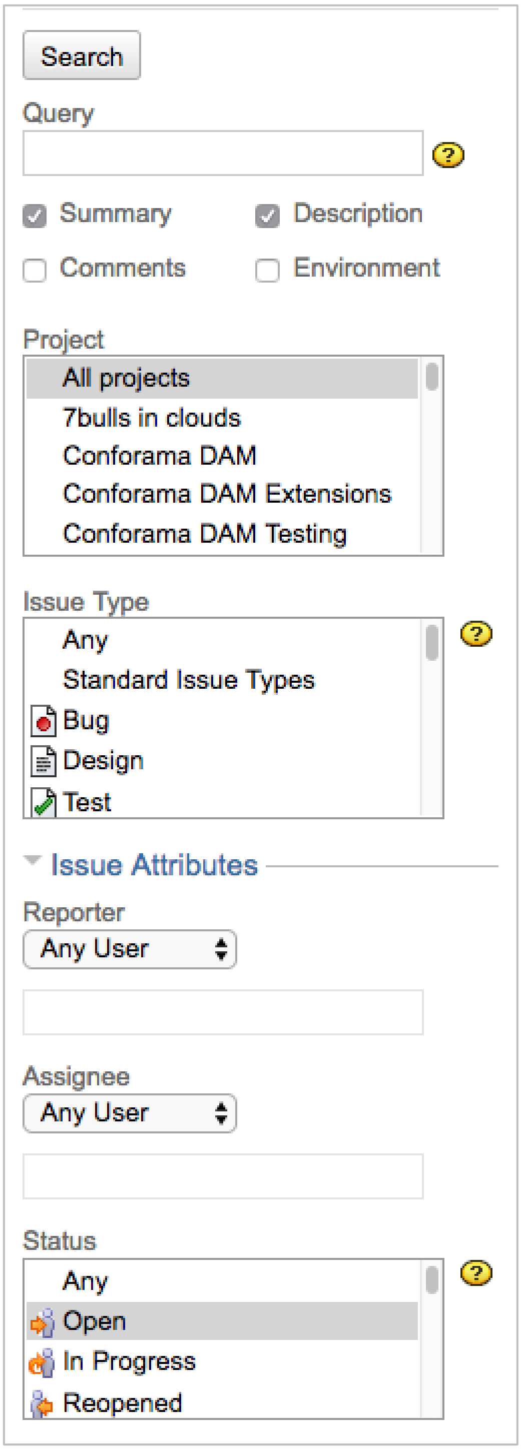


Figure 6 – JIRA Search Issue 2

   c. entering the appropriate search criteria (Figure 7) and press the 'Search' button

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664



Search

Query

☑ Summary ☑ Description

□ Comments □ Environment

Project
- All projects
- 7bulls in clouds
- Conforama DAM
- Conforama DAM Extensions
- Conforama DAM Testing

Issue Type
- Any
- Standard Issue Types
- Bug
- Design
- Test

Issue Attributes

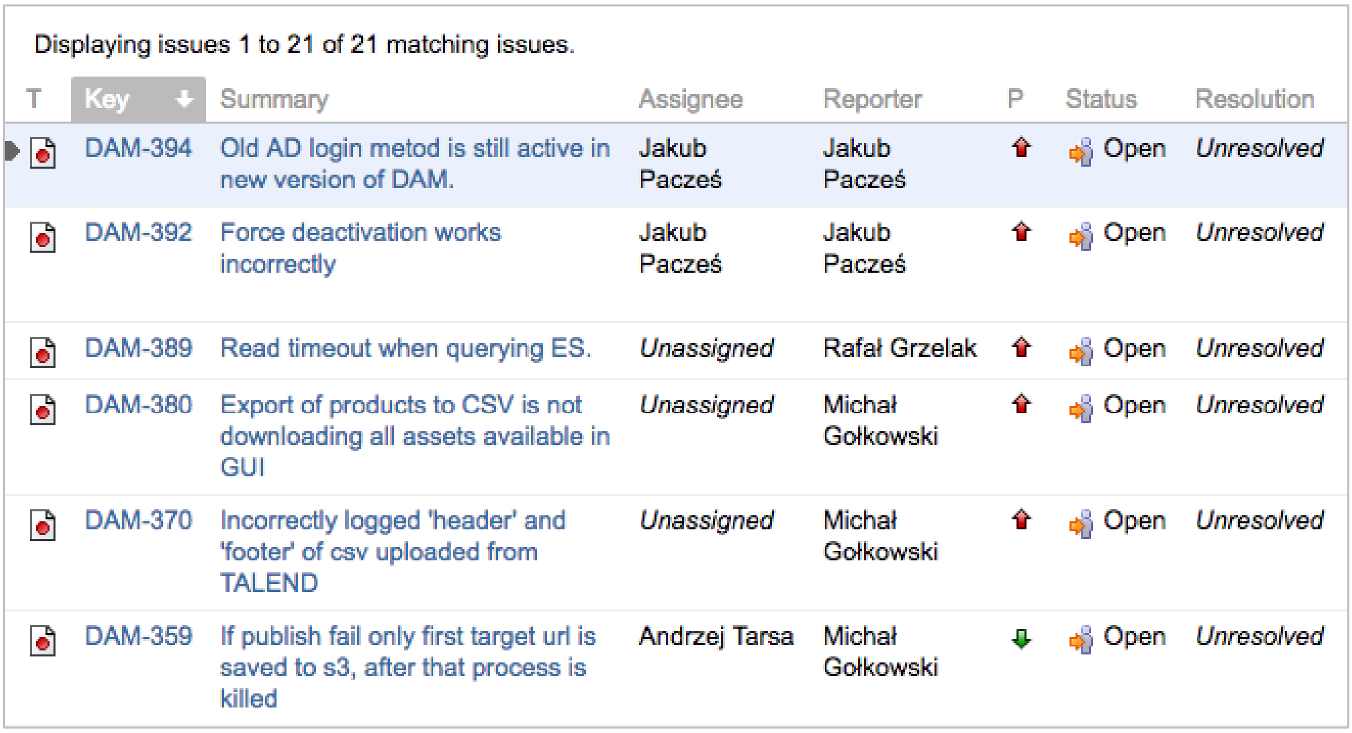
Reporter: Any User

Assignee: Any User

Status
- Any
- Open
- In Progress
- Reopened

Figure 7 – JIRA Search Issue 3

After following steps 2.a-2.c, you will get a list of appropriate issues. See Figure 8 for an example list.



Displaying issues 1 to 21 of 21 matching issues.

T	Key	Summary	Assignee	Reporter	P	Status	Resolution
	DAM-394	Old AD login metod is still active in new version of DAM.	Jakub Pacześ	Jakub Pacześ	↑	Open	*Unresolved*
	DAM-392	Force deactivation works incorrectly	Jakub Pacześ	Jakub Pacześ	↑	Open	*Unresolved*
	DAM-389	Read timeout when querying ES.	*Unassigned*	Rafał Grzelak	↑	Open	*Unresolved*
	DAM-380	Export of products to CSV is not downloading all assets available in GUI	*Unassigned*	Michał Gołkowski	↑	Open	*Unresolved*
	DAM-370	Incorrectly logged 'header' and 'footer' of csv uploaded from TALEND	*Unassigned*	Michał Gołkowski	↑	Open	*Unresolved*
	DAM-359	If publish fail only first target url is saved to s3, after that process is killed	Andrzej Tarsa	Michał Gołkowski	↓	Open	*Unresolved*

Figure 8 – JIRA Search Results

d. Then you can press one of the available listed issues

After executing case 1 or case 2, you will get to the details of the chosen bug and then you can edit them. Additional information about how to use JIRA can be found in the JIRA Users Guide[4].

3.4 Bugs life cycle

We will have the following states for a bug: "*Open*", "*Blocked*", "*In progress*", "*To test*", "*Closed*". Figure 9 depicts the management workflow for bugs in Melodic which is analyzed in more detail in the following sub-sections.

[4] https://confluence.atlassian.com/jira064/jira-user-s-guide-720416011.html

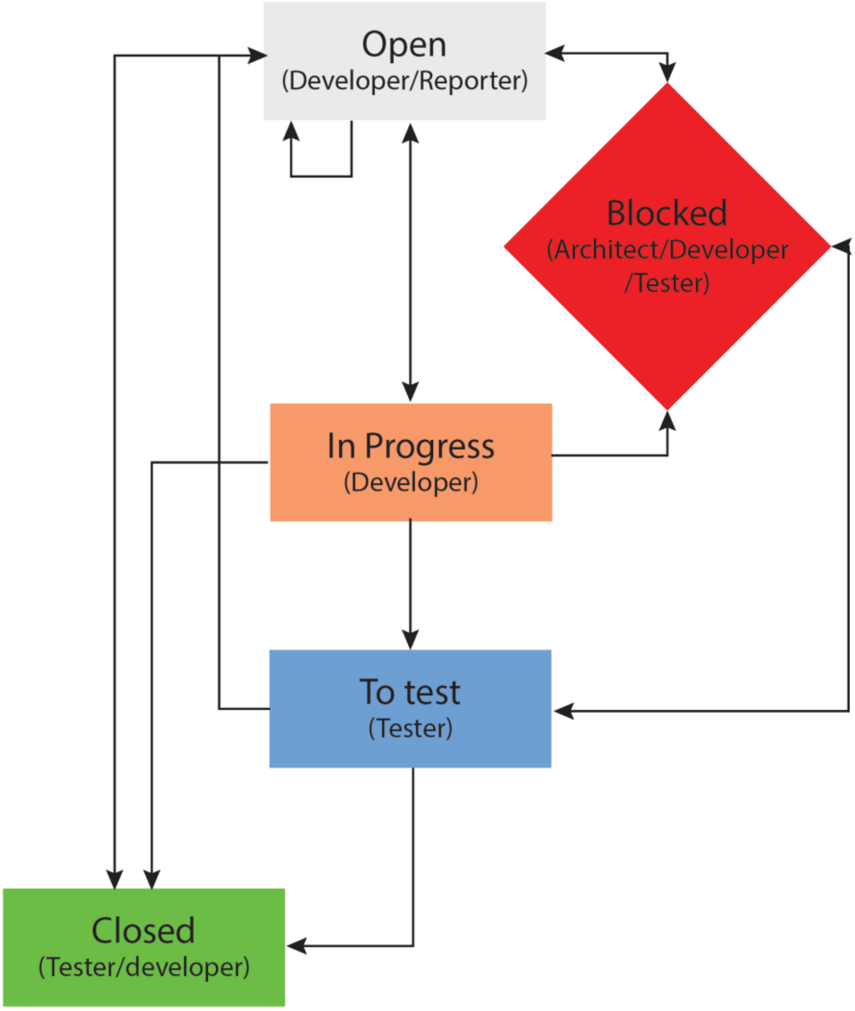


Figure 9 – Bug's management workflow

3.4.1 New issue (*'Open'* state)

After finding the bug, it will be reported in the JIRA system as described in chapter 3.3 - "Bug Reporting Process" with the status *'Open'*. The bug will be assigned to the Developer who has developed the respective part of the system. Before working on the issue, the developer will check if the information contained in the issue is sufficient and the error can be reproduced. This then maps to the following 4 cases:

1. If everything is clear and sufficient information has been provided for reproduction, the developer can start working on the bug. To this end, after the bug resolution starts, the developer will set the bug state to 'In progress'.
2. If the bug is incorrectly assigned to the developer who was not involved in a given part of the system, he will assign the bug to his reporter with the state *'Open'* and will add the desired information in the comment about the incorrect assignment.

This project has received funding from the European Union's Horizon 2020
research and innovation programme under grant agreement No 731664

3. If the developer needs some additional information about the way the system works, he will set the bug state to *'Blocked'* and will assign the bug to: (a) the respective Architect to explain what information should be delivered at the architectural level (examples of needed actions: explain and confirm interaction and communication between components, explain the role of the component, confirm the interface logic between components and so on), or (b) the Developer to provide additional information on how certain components should work (examples of needed actions: explain how the component is implemented, confirm the logic of how the particular interface works, provide representative values for fields/methods and so on). The assigned Architect/Developer will add missing information, change state to *'Open'* and assign the bug to the person from whom the issue was received.
4. If the developer needs some additional information about the bug, including the:
    a. environment
    b. version
    c. steps to reproduce
    d. expected result

    he will set the bug state to '*Blocked'* and will assign the bug to its reporter. The issue reporter will add missing or additional information, set the state to *'Open'* and assign the issue to the person from whom the bug was received.

3.4.2 Work in progress (*'In progress'* state)

During the whole process of bug resolution, the bug will have the state of *'In progress'* which has been set by the responsible developer. If one or more uncertainties appear, the developer will set the bug state to *'Blocked'* and assign the issue to:

1. Architect/Developer – if the developer needs some additional information about how the system works
2. Reporter of the issue – if the developer needs some additional information about the bug.

From the bug state 'In progress', the developer working on the bug can transition it to the following state alternatives:

1. *'Open'*
    a. When the developer stops working on the issue for a long time, but he will plan working on it in the future; the bug will be still assigned to the current developer.
    b. When the developer moves the bug to another developer; the bug will be assigned to the other developer in this case.
2. *'To test'*
    a. When the developer corrects the bug, the bug will be assigned to the test team:
        i. If the bug was created by a tester or a test leader, the developer will assign the bug to the reporter of the issue

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

ii. If the bug was created by a developer or an architect, the developer will assign the bug to the test leader who will then assign the bug to the appropriate tester.

3.4.3 Finishing the development (*'To test'* state)

After a bug is resolved, the developer will change its state to *'To test'* and assign the bug to the tester or test team. The developer will also add information about the versions of the software on which the bug is corrected. From the state *'To test',* the tester checking the bug can transition the bug to the following state:

1. *'Open',* if one of the following conditions hold:
    a. the bug has not been corrected
    b. the correction is different than described in the expected result
    c. the correction introduces another bug, very similar to the described bug
    d. missing information about the version number in which the bug will be corrected

    The bug will be assigned to the developer who corrected the bug.
2. *'Closed'*
    a. the bug is corrected – the received result is the same as the expected result, the bug will remain with status *'Closed'* assigned to the tester.

3.4.4 Release acceptance (*'Closed'* state)

The state *'Closed'* will be set in the following situations:

1. from the state *'To test'* - after executing tests on the modified software for this bug and the result is consistent with the expected result.
2. from the state *'Open', 'In progress'* if the bug is:
    a. obsolete;
        i. as the specific function is no longer present in the system
        ii. as the bug cannot be reproduce - a new version of the software has been produced and that version does not exhibit the bug any more
    b. a duplicate - the same problem has been reported by someone else in the JIRA system

The state *'Closed'* for the bug can only be set by its reporter. From the state '*Closed'* we will be able to re-open the bug in the following cases:

1. The bug reappears in one of the respective software newer versions
2. The bug reappears after executing regression tests

3.5 Bug's elements

This topic contains information about the bugs' elements in the Melodic project. A single bug will contain the following information partitioned into 3 categories: *Required, Optional, Automatically.* Information from the first two categories can be edited while the one from the last category is automatically created by the bug reporting system itself and is thus not editable:

3.5.1 Required

1. *Summary* – title, a brief description for a quick identification of the problem. The title should clearly and accurately describe what the bug concerns
2. *Environment* – the name of the environment in which the bug has been found and whether we can also reproduce the bug in other environments. If additional information about environment is needed, it will be added to the description of the bug in section "Reproduction steps" (see 4.a. below)
3. *Version number* – on which version the bug appears and whether we can also reproduce the bug on other versions
4. *Description of the bug*, which contains the following information:
    a. *Reproduction steps* – step-by-step process performed in order to obtain the bug
    b. *Current result* – the current state of the system after executing the reproduction steps
    c. *Expected result* – how the system should behave after executing the reproduction steps
5. *Status of the bug* – description in chapter 3.4.
6. *Priority of bug* – indicates the importance or urgency of fixing a bug, i.e., how important is the correction for the current bug and how fast it should be corrected
7. *Assignee* – to whom the bug is assigned
8. *Components* – additional information about the part of the system that the bug concerns
9. *Fix Version/s* – version number(s) of the affected software where the bug has been corrected. This information will be added by the developer upon submission of the issue to the test team.

3.5.2 Optional

1. *Initial conditions* – the state of the system needed to reproduce the bug, e.g. that the specified CAMEL configuration model has been provided and logged into the system, and that we are at the beginning of the CAMEL model processing, in the "Ready to profile" state
2. *Attachments*: logs, screens of incorrect situation, CAMEL model, etc.

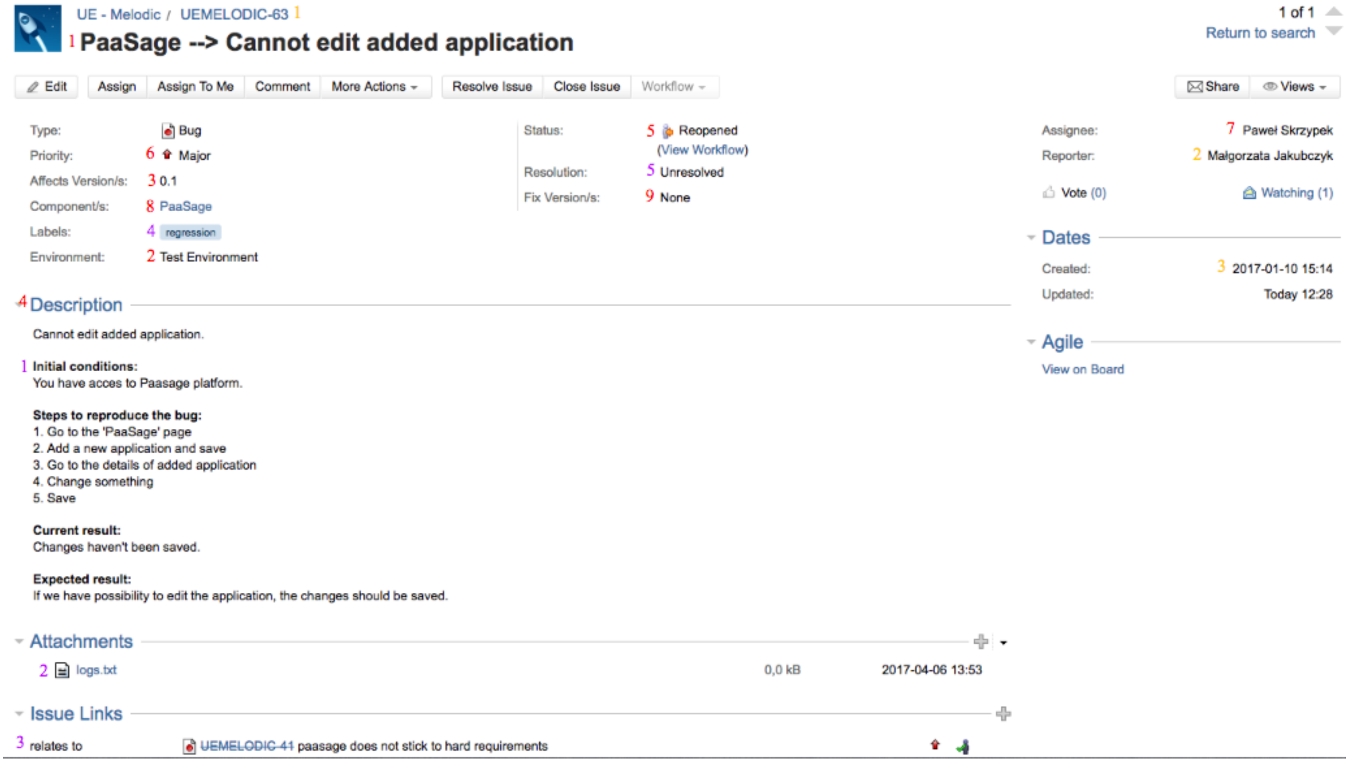
3. *Linked*[5] *issue* – related bugs or test cases, for instance: bug1 "causes" bug2 or test case1 "is blocked" by bug1, where "causes" and "is blocked" are the involved relationships, respectively.
4. *Labels* – to classify bugs, multiple labels can be provided for a single bug, see more details in section 3.7.
5. *Resolution* – information about the way the bug was resolved. This information will be added mainly by testers, but it can be added also by developers. The information can be added during issue closing.
6. *Test data* – relevant test data, needed to execute the test case, should be attached to the case. E.g. if a test case subject is to test deployment of the application described in file model.camel, the file model.camel should be attached to the Test Case. Also the steps needed to upload a file into the system should be described.

3.5.3 Automatically – elements added by the system

1. *Issue ID* – unique identifier created automatically when the bug is created
2. *Reporter* – the person who has created the bug
3. *Creation date* – the date of the bug creation

Required (red font), optional (purple font) and automatically filled (orange font) fields are marked on the screenshot in Figure 10.

[5] JIRA feature that allows you to create different types of links, allows flexible relationships between issues.



UE - Melodic / UEMELODIC-63 1

1 PaaSage --> Cannot edit added application

Type: Bug
Priority: 6 Major
Affects Version/s: 3 0.1
Component/s: 8 PaaSage
Labels: 4 regression
Environment: 2 Test Environment

Status: 5 Reopened (View Workflow)
Resolution: 5 Unresolved
Fix Version/s: 9 None

Assignee: 7 Paweł Skrzypek
Reporter: 2 Małgorzata Jakubczyk
Vote (0)
Watching (1)

Dates
Created: 3 2017-01-10 15:14
Updated: Today 12:28

Agile
View on Board

4 Description

Cannot edit added application.

1 **Initial conditions:**
You have acces to Paasage platform.

Steps to reproduce the bug:
1. Go to the 'PaaSage' page
2. Add a new application and save
3. Go to the details of added application
4. Change something
5. Save

Current result:
Changes haven't been saved.

Expected result:
If we have possibility to edit the application, the changes should be saved.

Attachments
2 logs.txt 0,0 kB 2017-04-06 13:53

Issue Links
3 relates to ~~UEMELODIC-41~~ paasage does not stick to hard requirements

Figure 10 – Bug's example

3.6 Bug priorities

The following bug priorities have been designated in the Melodic project, where such priorities are added by the bug reporter during bug opening: *Blocker, Critical, Major, Minor, Trivial*.

1. *Blocker* – related to data loss, missing feature, security violation, failure of the complete software system, subsystem or a program within the system; the bug should be resolved immediately
2. *Critical* – the bug does not cause a failure, but causes the system to produce incorrect, incomplete, inconsistent results or impairs the system usability; the bug should be resolved as soon as possible in the normal course of development activity
3. *Major* – a feature is missing but a workaround exists; the bug should be repaired after "blocker" and "critical" bugs have been fixed
4. *Minor* – a bug with minor impact on the system which does not prevent it from proper functioning; it can be resolved in a future major system revision
5. *Trivial* – a cosmetic error, a spelling mistake, etc.; it can also be resolved in a future major system revision.

3.7 Bug labels

This topic contains information about labels of bugs in the Melodic project. Labels are going to be exploited for the classification of bugs. Labels can be added to a bug at any stage in its management, even in the 'Close' status. New labels can be added by all team members, i.e., by Architects, Developers, Test Leader and Testers. The labels naming will be based on the following pattern:

1. Only small letters must be used
2. "_" must be used as a space between separate words.

Before adding a new label, it has to be verified whether such a label already exists or a very similar label has been created by someone else before. We provide below an extensible list of bug labels for the MEODIC project. The list can be extended with new labels anytime during the project lifetime.

- automated – the bug was detected during automated tests and is easy to reproduce.
- automated_new – the label indicates that a new automated test for the bug will be created. After this test is automated, this label will be removed by the creator of the automated test
- integration – the bug was detected during integration tests
- missing_information – more information about the problem is needed; the bug with this label will be assigned to its reporter
- performance – the bug was detected during performance tests
- regression_new – the bug was detected during regression tests
- regression_old – the bug was re-discovered in regression tests, it means the bug was corrected and closed, but has occurred again in a regression test
- security – the bug was detected during the security tests
- unit_tests – the bug was detected during unit tests
- duplicate – the bug is identical to another one, already created
- suspended – the bug resolution has been suspended
- obsolete – the affected part of the system has been changed (I.e., affected function not used any more or new software version no longer exhibiting the bug) so the error does not occur any more
- re-opened – a bug has been opened again, it has been found in newer versions of a software
- re-assigned – it will be used to denote that the bug was (re-)assigned to a different developer
- very_frequent – the bug occurs very often
- frequent – the bug occurs quite often
- rare – the bug rarely occurs

4 Test Case Management

This chapter contains detailed information about test cases in the Melodic project. In particular, it contains information about the test case lifecycle as well as the test case elements, types and priorities.

A test case is a set of test data, pre-conditions, expected results and post-conditions for the implementations, developed for a specific purpose or for the condition mapping to the test, such as the execution of a program path, or to verify compliance with a specific requirement. A test case describes how to perform a specific test. Test cases will be created by the test team, either its members or the test leader.

4.1 Audience

This chapter is intended for those involved in the software quality assurance process

- mandatory for test teams and architects:
  - the test team needs to know what is the test case creation process, what is the life cycle of the test case, and which elements the test case includes
  - architects have to check whether the test cases are consistent with the (system) specifications
- recommended for developers – they should know what is the life cycle of the test case and how the system will be tested
- optional for the rest of the project members.

4.2 Responsibility

Based on its current state, a test case will have to be handled by a specific role. Such responsibilities along with their mapping to respective roles are provided below:

Architects – handling of test cases with state 'More info needed'
Developers – handling of test cases with state 'More info needed'
Test team – handling of test cases with states 'New', 'Ready to run', 'In progress', 'Passed', 'Failed', 'More info needed', 'Close'.

Detailed information about the roles responsible to manage a test case based on its respective status is supplied in chapter 4.4 - "Test case life cycle".

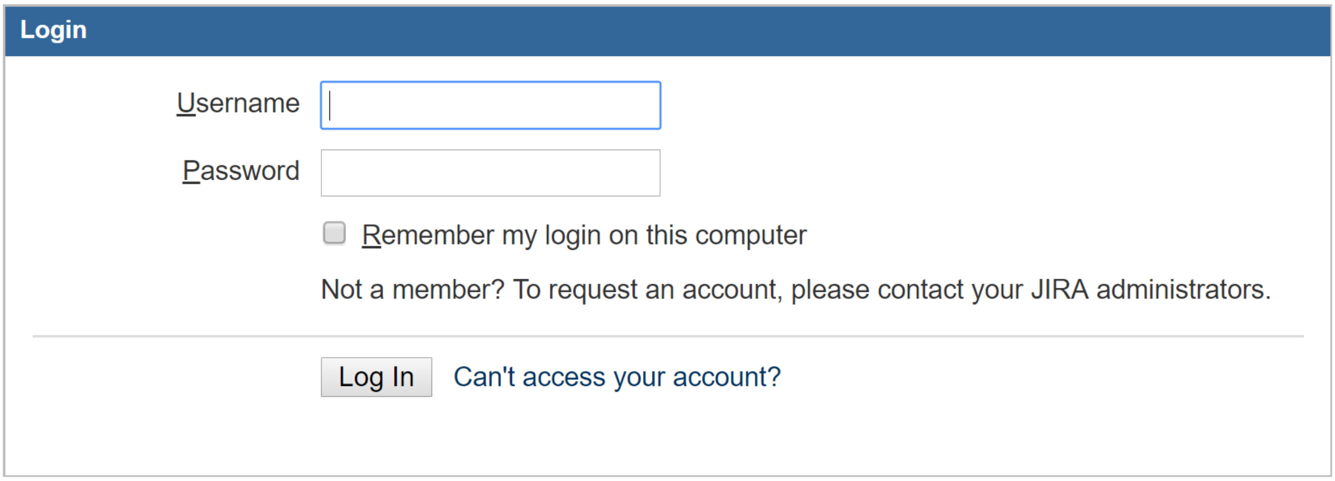
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

4.3 Test Case Creation Process

Test cases will be prepared in the JIRA[6] system for testing.

The process below details how to create a test case in the JIRA system:

1. In the browser, open the page: https://jira.7bulls.eu
2. Enter the correct values to the fields 'Username' and 'Password' and press the 'Log In' button:



Login

Username

Password

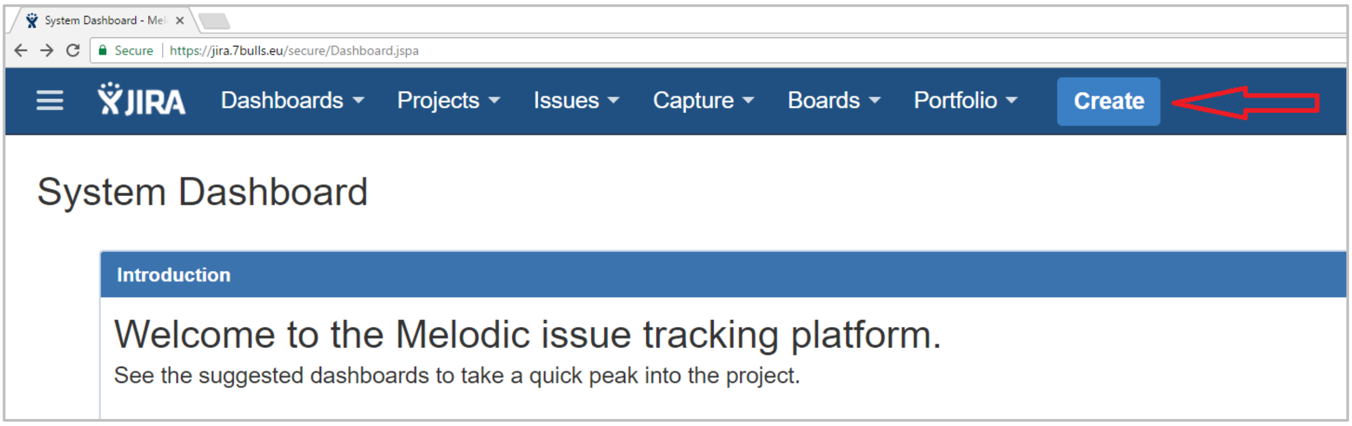
□ Remember my login on this computer

Not a member? To request an account, please contact your JIRA administrators.

Log In | Can't access your account?

Figure 11 – JIRA Login Page

3. To create a new test case, press 'Create':



System Dashboard

Introduction

Welcome to the Melodic issue tracking platform.

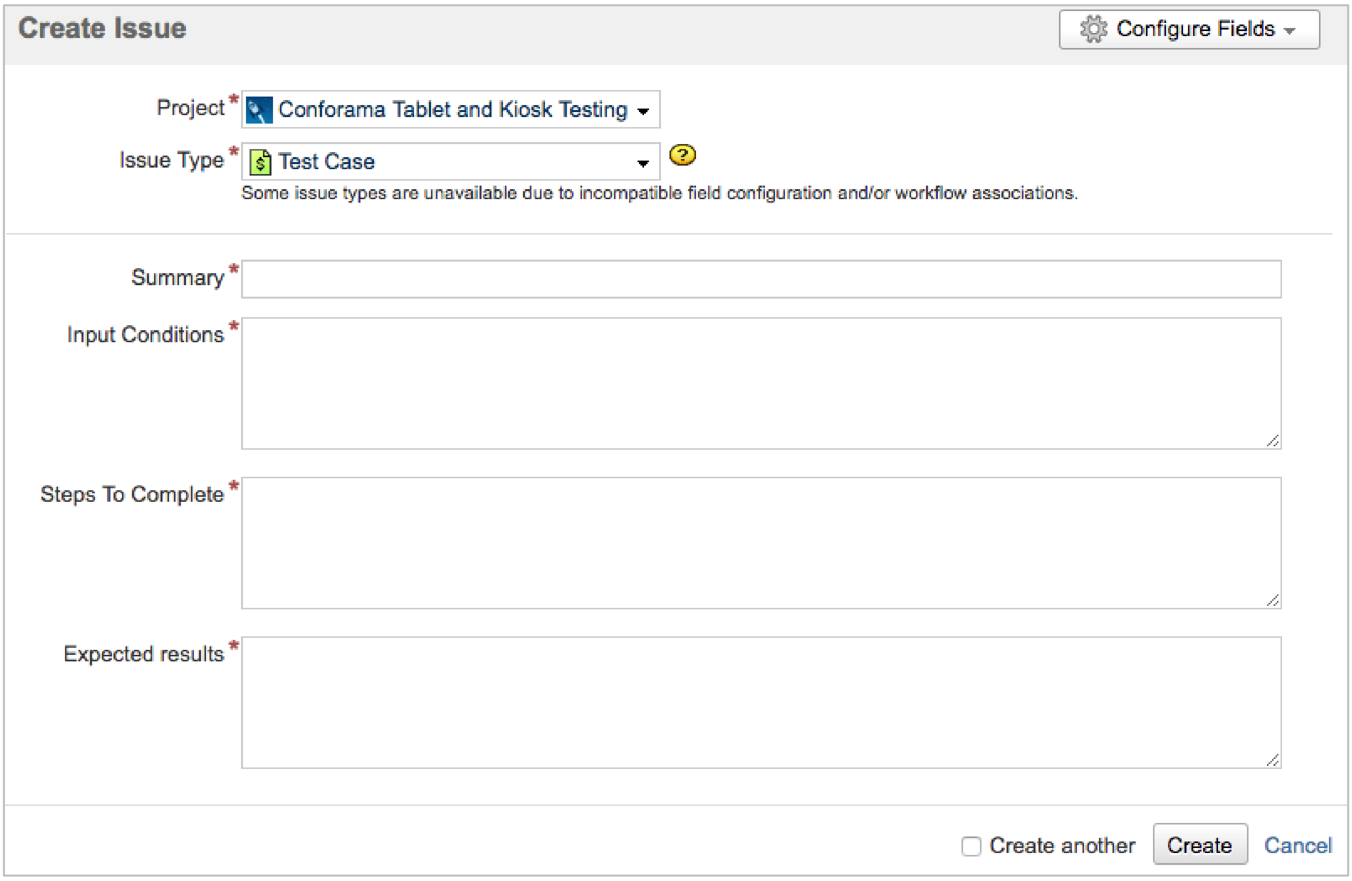
See the suggested dashboards to take a quick peak into the project.

Figure 12 – JIRA Home Page

4. Fill in all required fields and press the 'Create' button:

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

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664



Create Issue

Project* Conforama Tablet and Kiosk Testing

Issue Type* Test Case

Some issue types are unavailable due to incompatible field configuration and/or workflow associations.

Summary*

Input Conditions*

Steps To Complete*

Expected results*

□ Create another

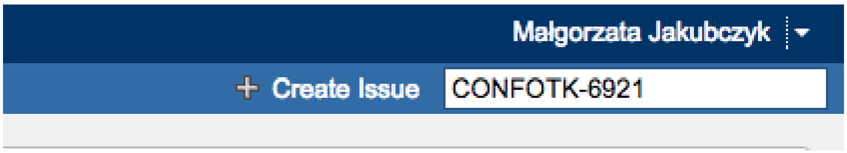
Figure 13 – Create Test Case

5. After following these steps, a new test case will be created.

Chapter 4.5 - "Test case elements" covers all the (required and optional) information that can be provided about a certain test case.

A test case can be identified and edited via the following process:

1. Enter the issue number in the search box and press the 'Enter' button:



CONFOTK-6921

Figure 14 – Search Test Case

or

2. Search the issue via:
    a. pressing the arrow next to the 'issue' button
    b. choosing the option "Search for issues"

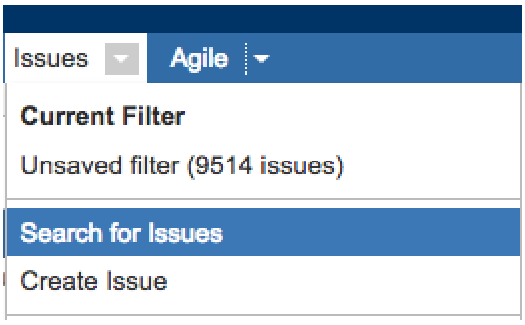
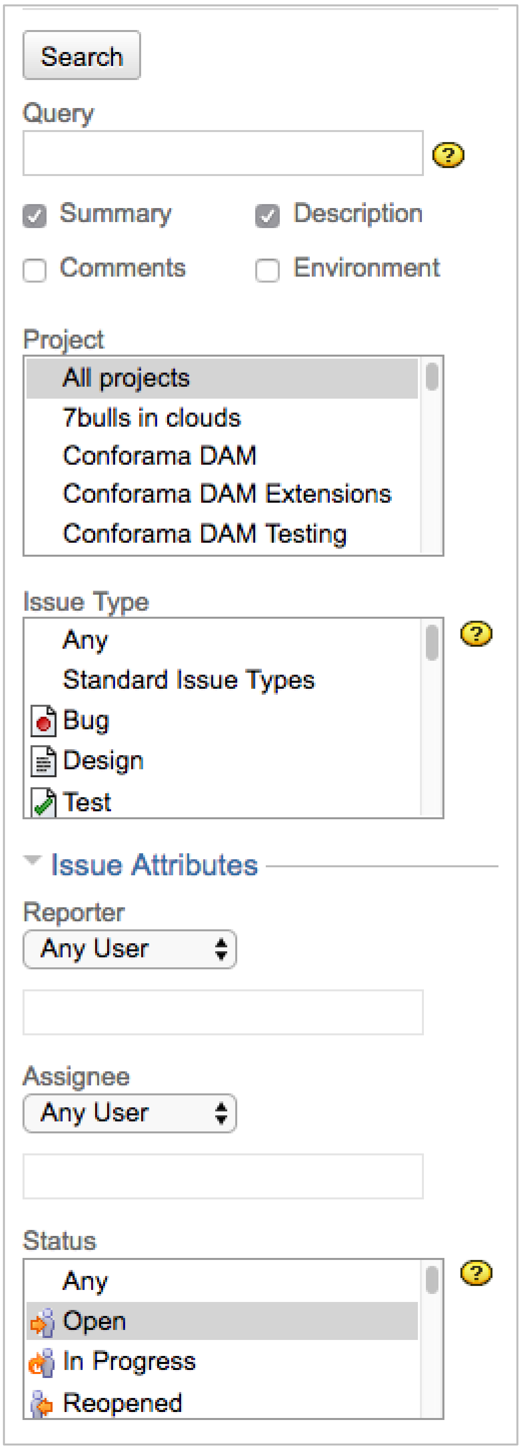


Figure 15 – Search Test Case 2

    c. entering the appropriate search criteria and pressing the 'Search' button (see Figure 16):



Search

Query

☑ Summary ☑ Description

□ Comments □ Environment

Project

- All projects
- 7bulls in clouds
- Conforama DAM
- Conforama DAM Extensions
- Conforama DAM Testing

Issue Type

- Any
- Standard Issue Types
- Bug
- Design
- Test

Issue Attributes

Reporter

Any User

Assignee

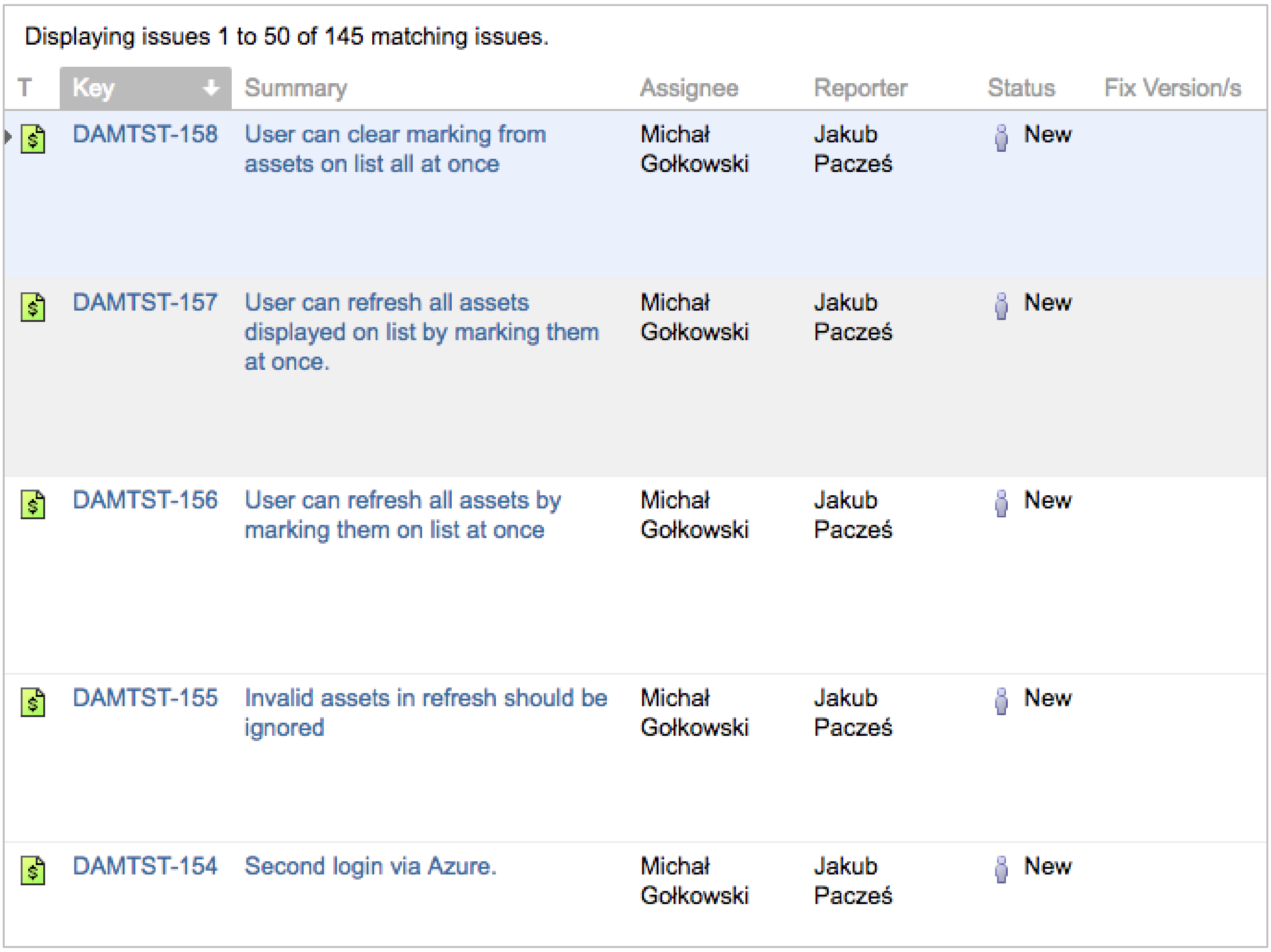
Any User

Status

- Any
- Open
- In Progress
- Reopened

Figure 16 – Search Results

After following steps 2.a-2.c, you will get a list of appropriate issues:



Displaying issues 1 to 50 of 145 matching issues.

T	Key	Summary	Assignee	Reporter	Status	Fix Version/s
	DAMTST-158	User can clear marking from assets on list all at once	Michał Gołkowski	Jakub Pacześ	New	
	DAMTST-157	User can refresh all assets displayed on list by marking them at once.	Michał Gołkowski	Jakub Pacześ	New	
	DAMTST-156	User can refresh all assets by marking them on list at once	Michał Gołkowski	Jakub Pacześ	New	
	DAMTST-155	Invalid assets in refresh should be ignored	Michał Gołkowski	Jakub Pacześ	New	
	DAMTST-154	Second login via Azure.	Michał Gołkowski	Jakub Pacześ	New	

Figure 17 – Search Results 2

e. Then you can press one of the available listed issues

After following step 1 or 2, you will be able to see and edit the details of a chosen test case. Additional information about the usage of JIRA for testing can be found in the JIRA Users Guide[7].

4.4 Test case life cycle

This chapter contains information about the test case lifecycle in the Melodic project. The following states apply for a certain test case: *"New"*, *"More info needed"*, *"Ready to run"*, *"In*

[7] https://confluence.atlassian.com/jira064/jira-user-s-guide-720416011.html

progress', *"Passed"*, *"Blocked"*, *"Failed"*, *"Closed"*. Figure 18 depicts the workflow for test case handling in Melodic.

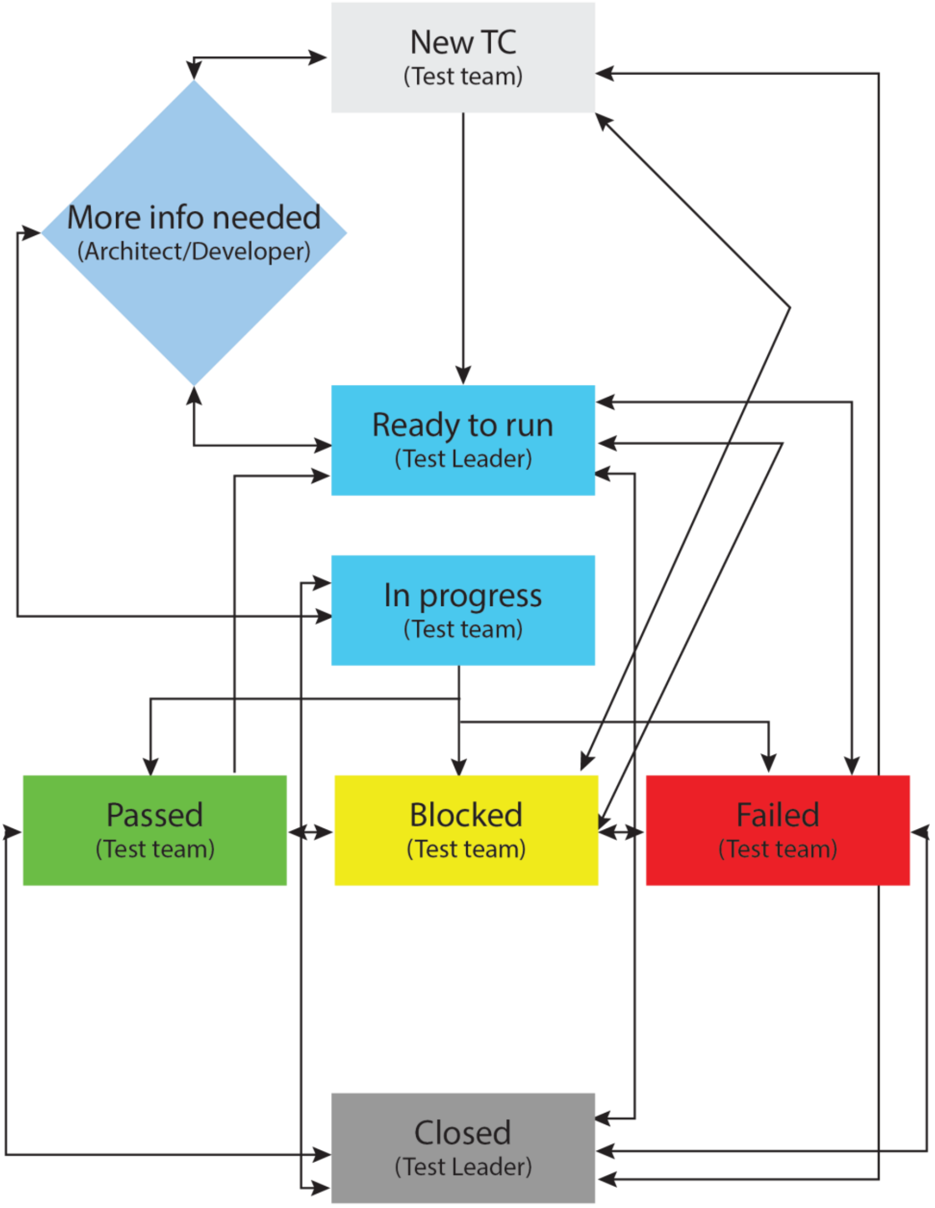


Figure 18 – Test case's workflow

After a test case is created by one of the members of the Test team, it will have the state of *'New'* and it will be assigned to the Test Leader. Then, the test case can transit to one of the following status labels:

1. If everything is clear in the test case, i.e., it contains all required information and the test case is ready to run, the Test Leader will set its state to *'Ready to run'*.

2. If the Test team needs some additional information about how the system works, it will set state to *'More info needed'* and assign the test case to the Architect or a Developer. The Architect/Developer will add missing information, change test case status to *'New'* again and assign the test case to the person from whom the question was received.
3. If the tested part of the system is blocked, Test team will be able to set state *'Blocked'*.

From test case state *'Ready to run'*, the Test Leader will be able to transit to one of the following states:

1. If everything is clear in the test case and the part of the system, which the test case checks, then it is ready for testing. The Test Leader will set the state to *'In progress'* and assign the test case to a member of the test team.
2. If the Test team needs some additional information about how the system works, it will set the test case state to *'More info needed'* and will assign the test case to the Architect or Developer. Architect/Developer will add missing information and change the state to *'Ready to run'*.
3. If the part of the system cannot be tested, the test team will have set the *'Blocked'* state.

From state *'In progress'*, the test team will be able to transit to one of the following states:

1. if everything is clear, the respective testers will execute the corresponding test case. After checking the test case, the tester can transit to the following states:
    a. *Passed*:
        i. system works according to the specification
        ii. received result is the same as expected result
        iii. all steps passed with status OK

        All conditions/points (i, ii, iii) must be satisfied to mark the test case as positive (passed).

    b. *Failed:*
        i. system does not work
        ii. system works differently with respect to its specification
        iii. received result is different than the expected result
        iv. one or more steps have status KO (not ok) .

        If any from the above points (i, ii, iii, iv) is satisfied, we will mark the test case as negative (failed).

    c. *Blocked:*
        i. one or more steps are blocked by a bug
        ii. checking a part of the system that is blocked by another part of the system which does not work

        If any from the above points (i, ii) is satisfied, we will mark the test case as blocked.

Test cases with states '*Blocked*' and '*Failed*' will be retested.

2. if not everything is clear or Test team needs some additional information about the way the system works, Test team will set the state to '*More info needed*' and will assign the test case to the Architect or Developer. The Architect/Developer will provide the missing information and the state will be transitioned to '*In progress*'.

From states '*Passed*', '*Blocked*', '*Failed*' we will have to set state '*Ready to run*' if the test case needs to be retested or we need to perform regression tests.

A test case will be able to have its state set to '*Closed*' in the following cases:

1. test case is obsolete - part of system has been changed, so the test case is no longer useful
2. the concerned part of the system has been tested fully with positive result.

From state '*Closed*', the Test Leader can reopen the test case and set its state as '*New*' if the concerned part of the system has been changed and the test case must be verified again.
From the state '*Closed*', the Test Leader will have to reopen the test case and set its status as '*Ready to run*' if the test case must be run within regression tests. We will be able to set the state 'Closed' from all states except for the '*More info needed*' one - in that state all ambiguities should be clarified and passed to the test team for acceptance.

4.5 Test case elements

This section supplies information about the elements of a test case in the Melodic project. A single test case will contain the following information pieces, classified according to their category:

4.5.1 Required

1. *Summary* – briefly describes what the test case will do
2. *Preconditions* – conditions that must be fulfilled before the component or system can be executed with a particular test case; the preconditions will be provided in a textual form, e.g., you have prepared the 'Camel.xml' file, you enter it into the system, so you are in the state 'Ready to profile'.
3. *Test data* – the data necessary to carry out the test; example of test data: if we test adding applications specified in CAMEL, a Camel.xmi file is a sample of test data, i.e., the test data maps to the CAMEL Camel.xmi file which includes the description of the applications to be added.
4. *Steps to execute* – steps that users must take to run a test case
5. *Expected result* – the desired result after following these steps
6. *Priority of test case* – how important the test case is; more information can be found in chapter 4.7 and deliverable “D5.06 Test Strategy and Environment”

7. *Assignee* – to whom the test case is assigned; who will execute the test case or who will add missing information to it
8. *Status of the test case* – details already provided in chapter 4.4 - "Test case life cycle"
9. *Components* – additional information about which part of the system the test case concerns
10. *Environment* – information about the environment on which the test case will be or was executed,
11. *Version number* – after testing the test case, information about the version of the system or component on which the test case was executed will be added

4.5.2 Optional

1. *Linked*[8] *issue* – related user story, bugs or test cases; for instance: test case1 is testing user story1, test case1 has failed due to bug1, test case2 is blocked by bug2 where "is testing", "has failed" and "is blocked" are the involved relationships, respectively.
2. *Attachments* – logs, screens from the result of test case, etc.

4.5.3 Automatically – elements added automatically by JIRA

1. *Test case ID* – unique identifier created automatically when the test case is created
2. *Reporter* – the person who created the test case
3. *Creation date* – the date of creation of the test case.

4.6 Test case types

The JIRA system gives possibilities to add new fields. Such kind of a new field will be added for test cases. The field will be named 'type'. The following types of test cases will be considered in the Melodic project:

1. *Smoked*:
   After each deployed build, this sort of quick check will be performed. Quick check means checking all basic functionalities. The quick check will be executed by the tester.
2. *Automated*:
   Progress will be tested on a regular basis - automatic test cases will be prepared and run after deploying and delivering the functionalities to test.

[8] JIRA feature that allows you to create different types of links, allows flexible relationships between issues.

3. *Manually*:
   As with automated tests, manually performed test cases will be run after putting the functionalities to test.
4. *Automated regression*:
   Automated test cases will be performed after each big release. We will have 3 big releases: M12, M24, M36.
5. *Manual regression*:
   Manual test cases will also be performed after each big release.

Test Leader will have the possibilities to add new fields for Test Cases at any stage of the project if additional information of test cases is needed.

4.7 Test case priorities

The following test case priorities have been designated in the Melodic project, where such priorities are set by the test case's reporter during its creation: *Critical, Major* and *Minor*.

All test cases with priorities 'Critical' and 'Major' must be executed and finished with status 'Passed' before each major release in the Melodic project.

More detailed information about test case priorities are to be found in "D5.06 Test Strategy and Environment".

5 Testing tools

For Unit Testing described in D5.06, the following tools are recommended:

5.1 JUnit

JUnit is a java test framework to write repeatable tests. It encourages developers to write single unit tests as early as possible in the development phase. Tests are written just like Java methods with the use of special framework annotations. Download and installation instructions for the tool can be found on the tool's Internet page, as well as a complete Junit tutorial [9, 10].

5.2 Spock

Spock is another unit testing framework for Java and Groovy applications which utilizes Groovy's syntax for test creation. The Spock test specification language was created based on past experience with JUnit, JMock, Groovy and other frameworks and languages. This leads to a pretty solid product which promises to write tests in less time and with bigger test clarity (thanks to the separation of test logic from the method logic itself). Single page documentation is in the tutorial[11], while download and installation instructions of the tool you can find on the tool's Internet page[12].

5.3 TestNG

TestNG is a very popular Java automated testing framework. It is similar to Junit, but it is not a Junit extension – it is more like an evolution of it. It should be fairly easy to migrate from JUnit to TestNG to take the advantage of all the features introduced by TestNG. A quick comparison

[9] http://spockframework.org/spock/docs/1.1-rc-3/all_in_one.html#_getting_started

[10] https://github.com/junit-team/junit4/wiki/Download-and-Install

[11] https://github.com/spockframework

[12] https://www.mkyong.com/unittest/junit-4-vs-testng-comparison/

between those two frameworks can be found here[13] and a getting started tutorial for TestNG can also be found on the tool's Internet page[14]. Download and installation instructions in Eclipse are also available[15].

5.4 Googletest

For components written in C++, the most suitable solution for unit testing would be the Google Test framework. It can be deployed on Linux, Windows, Mac OS and Cygwin environments and easily integrates with the Visual Studio. It can be used with the Google Mock extension to prepare and perform automated and repeatable tests. A quick start tutorial can be found at JetBrains[16].

5.5 SoapUI

Functional Testing, as described in "D5.06 Test Strategy and Environment", can be supported with SOAPUI, a very popular open-source tool for testing Web Services. It supports both SOAP and RESTful Web Services and can be installed on Linux, Mac or Windows. There are already some lucid step-by-step tutorials covering the following topics:

- functional testing[17]
- load testing[18]
- security testing[19]
- mocking with SoapUI[20]

Download[21] and installation[22] pages you can find on the Internet.

[13] https://www.tutorialspoint.com/testng/index.htm
[14] http://testng.org/doc/download.html
[15] http://qatechhub.com/installing-testng/
[16] https://blog.jetbrains.com/rscpp/unit-testing-google-test/
[17] https://www.soapui.org/functional-testing/getting-started.html
[18] https://www.soapui.org/load-testing/getting-started.html
[19] https://www.soapui.org/security-testing/getting-started.html
[20] https://www.soapui.org/soap-mocking/getting-started.html
[21] https://www.soapui.org/downloads/soapui.html
[22] https://www.soapui.org/getting-started/installing-soapui.html

5.6 JMeter (Apache)

For Load and Performance Testing, JMeter is recommended. The software has been developed by Apache Software Foundation on an open-source basis as a pure Java application. It is designed to measure and analyze the performance of applications by allowing to build and execute test plans. JMeter simulates a group of users sending requests to a target server and aggregates responses into statistics presented in tables/graphs. One of the advantages of using JMeter is the ability to set up a Load test in a distributed manner (using JMeter Server and number of injectors). Although there is quite solid documentation of JMeter on the JMeter site[23], it might be handy to start with a small tutorial[24]. Download and installation pages for this tool are also available[25].

5.7 Dependency-Check

Security Tests can be performed with the use of the above described SoapUI and with the OWASP Dependency-Check. It is a software utility that can identify project dependencies and checks if there are any known, publicly disclosed, vulnerabilities. For that moment it can be officially used to scan Java and .NET applications, but there are also some attempts to support Python, Ruby, PHP and Node.js applications. The result of a dependency check is presented in the clear form of a report containing information of vulnerable dependencies with an appropriate severity level. The utility can be used as a standalone command line tool, ant task or plugin (Gradle, Jenkins, Maven or SBT). Download and installation of the tool you can find on the tool's Internet page[26].

[23] http://jmeter.apache.org/usermanual/get-started.html

[24] https://www.tutorialspoint.com/jmeter/jmeter_quick_guide.htm

[25] http://jmeter.apache.org/download_jmeter.cgi

[26] https://jeremylong.github.io/DependencyCheck/index.html

6 Delivery management

In this chapter, we outline procedures to ensure effective release management as well as measures to assure high quality software delivery in the Melodic project. The platform deployments in Melodic will be automated through a dedicated continuous integration (CI) platform that will allow continuous building and integrating of the MELDOIC framework components and the use case applications. However, in cases where an automatic deployment fails, a manual deployment will be performed by respective responsible person, following a well-defined manual deployment process. Further details on the continuous integration will be supplied in the dedicated deliverable 'D5.05 Continuous integration platform & guidelines 'due in the second project year.

This chapter is intended for those involved in the development:

- mandatory for all developers, release managers, and the test team
- optional for all

6.1 Roles and Responsibility

We identify three main roles in the software delivery lifecycle: Developers, Release Manager, and the Test Team. Their responsibilities are listed below.

- Developers – prepare and deploy a new version of the system in Development and Integration Tests Environments
- Release Manager – responsible for deploying a new version on Acceptance Tests Environments
- Test team – the responsibility for testing the new version of system on Integration and Acceptance Tests Environments

More information about environments can be found in Chapter 2 "Software deployment environments".

6.2 Software Quality Assurance

In order to ensure high software quality, we have devised strict criteria to be met before the code is promoted to different test environments - until it reaches final acceptance. The measures for assuring the quality of the delivered software and the criteria for promoting code within environments is listed in the following:

1. For migration to the Integration Environment:
   To promote system code to the Integration Environment, the following criteria needs to be fulfilled:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

   a. Code review of the corresponding pull request should be done by at least one *code reviewer* from a different participant organisation than the developer. Responsibility to ensure that the code review is completed lies on the developer.
   b. All ten rules of Better Code Hub[27] should be fulfilled.
   c. The preparation and successful execution of the unit tests for all the delivered code is required. Minimum coverage of unit tests should be 40%, counted using prepared Sonarqube[28] rules. Again, the responsibility to ensure that the test cases are written is on the corresponding developer. The value of minimum coverage is selected based on the experience of the main software development partner, 7Bulls, in commercial software development projects.
   d. The test team will review the unit tests, suggest improvements, and ask the development teams to add more tests when needed.
2. For migration to the Acceptance Environment:
   To promote the system code to the Acceptance Environment, the criteria listed below should be fulfilled:
   a. The system should work as per the expectations with all components properly integrated and functioning. The is the responsibility of the architect and the developers who delivered the code for release.
   b. All integration tests should be prepared and successfully executed. Minimum combining coverage of unit and integration tests should be 50%, counted using prepared Sonarqube. Preparation and execution of the integration tests is the responsibility of the corresponding developer.
   c. Smoke tests should be executed with positive result. The execution of the smoke tests is the responsibility of the test team, while fixing discovered bugs are to be handled by the developers.
3. For migration to the Final Acceptance Environment:
   To promote the system code to the final acceptance environment, all the criteria listed in following should be fulfilled. Note that some of these criteria will be checked via automatic verification system while other need a manual verification check.
   a. All test cases related to the Melodic framework with priority set as *critical* or *major* must be passed.
   b. All non-functional requirements related to performance and security must be fulfilled and tested positively.
   c. All test cases related to the core functional flow of the Melodic platform must be passed.

[27] https://bettercodehub.com/

[28] https://www.sonarqube.org – tool for verification of the code quality, including unit/integration test coverage

d. The optimisation of the deployment architecture should be verified in selected scenarios.
e. Evaluation and validation of delivered features by use case partners is needed before the final acceptance.
f. The use case applications should be installed and tested on the new Melodic software, and all major features of these applications should work as per the expectations.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

7 Meetings

This chapter provides a summary of planned Quality Assurance meetings to be held in the Melodic project. The chapter is mandatory for all participants of the project. The 7bulls test team will organise regular/interval meetings to ensure that the software testing and quality assurance procedures are in place. Before each meeting, a doodle will be sent to the participants to get their availability and choose the most appropriate date and time slot (expect for the regular weekly meetings which will be held at fixed times every week). All meetings will take place on Zoom[29].

- *Regular weekly Meetings*
  There will be regular weekly meetings to discuss current bugs, problems, and determining who will fix/improve them and when. The timeslot for the weekly meetings will be chosen after consultation with the relevant development partners.
- *Review meetings for the test cases*
  After the preparation of a test plan is completed, the plan will be sent to all teams contributing in the Melodic project. After all teams have finished reviewing the document, 7bulls will organize a meeting to get feedback from all participants and discuss any remaining issues and problems.
- *Discussions on result of unit and integration tests*
  Once the execution of the unit and integration tests is completed, 7bulls will organize a meeting and discuss the result of these tests with the relevant developers.
- *DEMO after each iteration*
  We will have three major releases in the project. After each major iteration, we will prepare a DEMO to present the current realised system.

Following are the required members in the meetings:

- The Quality Assurance team of 7bulls
- The chief architect
- At least one developer from each development team.

[29] https://zoom.us/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

8 Summary

The 'Quality Assurance Guide' deliverable contains the approach to testing, bug management and software quality assurance in the Melodic project. In the document the following information has been described:

1. *The environments which will be utilised in this project*
   We will be using three environments in Melodic project; Development Tests, Integration Tests and Acceptance Tests environments. Development Tests environment will be used by Developers, Integration Tests environment will be used by both Developers and Test team, and Acceptance Tests environment will be utilised by the Test team.
2. *The processes of bug management*
   Bugs will be reported in the JIRA system according to the routines described with respect to the process of bug issue creations and handing, as well as information about bug elements, priorities and labels.
3. *The processes of test case management*
   Test cases, just like bugs, will be reported in the JIRA system. Information about the process of test case creation and handing, and information about test case elements, priorities and types are detailed.
4. *Testing tools which will be used in the project*
   For Unit Testing we will be using the following tools: JUnit, Spock, TestNG and Googletest. For Load and Performance Testing we will be using JMeter, for Security Tests we will be using OWASP Dependency-Check, and the Functional Testing will be supported by SoapUI.
5. *Delivery management and meetings*
   Information about delivery management and meetings which will take place during the project is detailed. The following testing meetings in the Melodic project will be held: *Review of test cases, Discussions on the result of unit and integration tests, Weekly meeting* and *DEMO* after each iteration. All meetings will be organized by 7bulls team.