

This is the Author Accepted Manuscript.

There may be differences between this version and the published version. You are advised to consult the publisher's version if you wish to cite from it.

[https://eprints.gla.ac.uk/235881/](https://eprints.gla.ac.uk/235881/)

Deposited on: 5 March 2021
Highlights

- Portrays the design and implementation of Sciit a novel distributed issue tracking system.
- Demonstrates the advantages of using Sciit to minimise friction in issue tracking during software evolution.
- Compares Sciit to state of the art distributed issue trackers.
- Incorporates a migration path from centralised issue trackers, such as Gitlab.
1. Introduction

The issue tracker has become the de facto means of coordinating work effort in agile software projects. In a typical workflow [1], tasks are logged as issues, small self contained packages of work. These may then be further elaborated by the project team and annotated with additional metadata, such as type (feature, bug etc.), priority, milestone, estimated effort and affected project components. Issues are then assigned to a developer for resolution. The developer works on the necessary changes, gradually making commits to a branch in the project’s source configuration management system (SCM). Eventually, the work is completed and the developer requests for the changes to be reviewed within the team’s quality assurance (QA) process. Once approved, the changes are merged from the feature branch into the main line.
of development, via a merge or pull request. Once these changes are accepted the issue status can be manually changed to resolved and the issue archived.

Reviewing this example workflow reveals that software developers are required to manually maintain the consistency of information between their SCM and issue tracker. Curating this information accurately can substantially increase the probability of issue resolution [2] as well as reduce the cost. Unfortunately, this requires additional effort that may lead to information in different systems that is incorrect, incomplete, inconsistent or out of date [3].

For example, when work begins on an issue, a developer must create a new branch in the SCM and the issue must be marked as ‘In progress’ in the issue tracker. When the developer completes a part of the resolution to the issue, they may make a commit of the new source code to the project’s SCM. They must then separately record a comment on the issue to report the progress made. When an issue moves to QA, the developer must remember to mark it as ‘in QA’ in the issue tracker, then create a merge request to the QA branch. Finally, when a feature reaches production, the developer must also remember to change the issue status to ‘Closed’ in the issue tracker. Aranda and Venolia [4]’s study of the history of ten defect reports demonstrated that a substantial amount of information about a defect’s resolution was not recorded in the formal issue tracking system. This suggests that the friction involved in recording progress on issue resolution is causing substantial amounts of potentially useful information to be lost. Anvik et al. [5] also noted the difficulty of relying on manually assigned metadata in issue trackers, since the practice of completing and maintaining the data (and the exact use of different fields) varies from project to project.

Fundamentally, the redundancy arises because of the artificial separation of issue tracking from source control management. We therefore argue that issues are early stage representations of software development work, that have strong dependencies on artefacts within the SCM and will eventually be represented through alterations to artefacts within the SCM. Issues should themselves therefore be treated as first class configuration items. All the changes necessary to realise the feature can be tracked in the SCM as change sets with associated commit messages. Storing the description of the work to be done as an early representation of the consequent implementation allows the associations between the two and their concurrently developing histories to be explicitly, rather than incidentally, linked. Many of the items of project metadata, such as issue status can also be inferred from the SCM history,
This paper is structured as follows. Section 2 presents the workflow for interacting with Sciit and illustrates how many aspects of redundancy in the software project management workflow can be eliminated by treating issues as source control items within the SCM. Section 3 presents the design and implementation of the Sciit tool and its integration with the Git and Gitlab packages. Section 5 compares Sciit with other distributed issue trackers and Section 6 summarises our conclusions and outlines future work.

2. Demonstration

In this section, we present a scenario to illustrate a typical workflow using Sciit. During the explanation, we show how many of the tasks required of a developer when working with an issue tracker are redundant when the issues are managed in the SCM; and further how much of the pertinent information about an issue can be obtained through analysis of the contents of the SCM. The chosen workflow, adapted from feature branching as in the Github workflow [1], is illustrative of how Sciit can be used to track issue

---

1https://gitlab.com/sciit/sciit
The file `demonstration.sh` in the project source repository for sciit can be inspected. The issue in the markdown file (Figure 1b) contains a title, description, assignees, due date, labels, weight and priority are all specified by the product owner. The full list of user created fields that can be included in Sciit are summarised in Table 1.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@issue</td>
<td>The unique ID of the issue in the repository (required)</td>
</tr>
<tr>
<td>@title</td>
<td>A single line short description of the issue</td>
</tr>
<tr>
<td>@description</td>
<td>A multi-line summary of work to be done</td>
</tr>
<tr>
<td>@assignees</td>
<td>A comma separated list of usernames</td>
</tr>
<tr>
<td>@due_date</td>
<td>The date the issue is scheduled for completion</td>
</tr>
<tr>
<td>@labels</td>
<td>A comma separated list of label strings</td>
</tr>
<tr>
<td>@weight</td>
<td>An estimate of the cost of the work to be done</td>
</tr>
<tr>
<td>@priority</td>
<td>The importance of the issue</td>
</tr>
<tr>
<td>@blockers</td>
<td>A comma separated list of issue ids</td>
</tr>
</tbody>
</table>

Table 1: Summary of user specified fields in Sciit issues

status. Issue status is determined within a single property method in the Issue class, `Issue.status`, and can therefore be readily replaced to fit other workflows.

The file `demonstration.sh` in the project source repository for sciit can be used to create a demonstration git repository and executes the commands necessary to follow the example scenario. In addition, a repository on Gitlab has been configured for the integration service so that the relationship between commits in a Git repository, Sciit issues and Gitlab issues can be inspected.²

In the scenario, a product owner in a Scrum team has just completed a user story workshop with key customers for an existing web application, resulting in a request for a new feature to be added to allow customers to attach a photograph of damage as evidence when making an insurance claim. Following the workshop, the product owner creates a new issue using the Gitlab issue tracker interface (Figure 1a). Sciit detects this and creates a new branch for the issue in the project repository on the server, creates a new markdown file in the `backlog` directory of the project repository source (as shown in Figure 1b), adds the file to the index and performs a commit.

The issue in the markdown file (Figure 1b) contains a title, description and priority, all specified by the product owner. The full list of user created fields that can be included in Sciit are summarised in Table 1. Title, description, assignees, due date, labels, weight and priority are

synchronised with Gitlab. In addition, the new issue command assigns an issue id for the issue, photo-upload-on-claim. Sciit ensures that all issues in a commit have a unique ID and rejects any commits where two issues in the code base have the same ID. In addition, the new issue command gives the same name, in this case photo-upload-on-claim, to the feature branch to assist with workflow tracking.

Later, a software developer pulls updates from the server to their local repository and reviews the new issue received. Figure 1c shows the new issue on the command line user interface. Notice that the summary presented by Sciit contains extra information about the issue, such as the issue participants, that is not present in the text file itself, but instead inferred from the version control system metadata. The issue status is also reported as Open (proposed) because Sciit interprets the presence of an issue in a feature
branch, but not in master, as having not yet been accepted into the project. When the product owner is satisfied with the issue content they can create a merge request for it. Once the merge is accepted, Sciit automatically reports that the issue status is now Open (accepted) and adds the identity of the developer who accepted the commit along with a timestamp to the issue information.

The developer then selects the feature for analysis and implementation. The developer identifies three tasks that must be completed in order to resolve the issue: updating the database schema, modifying a view to include uploading a photo on a claim and extending the user acceptance testing (UAT) suite. The main issue was created as a markdown file in the repository. However, issues can also be stored within source files as code comments anywhere in the repository. For example, the UAT sub-issue is created as a comment within the existing Gherkin features/claims.feature file, as shown in Figure 2a. A summary of supported languages and comment format is presented in Appendix B. Developers using Sciit can therefore place issues as close to their resolution in the project directory hierarchy as desired. The location of the issue is shown to the developer in the Sciit interface, giving the user information about where the work will need to be done to resolve the issue. Reviewing the status of the issues in Sciit (Figure 2b) the developer can now see that the parent issue has three children, all of which are open.

The developer works gradually on the UAT task. As soon as a commit is made to the branch for the issue it becomes marked by Sciit as in progress.
For example, the developer adds a first scenario, to the `claims.feature` file and then makes a commit, as shown in Figure 3a. This causes Sciit to automatically report the sub-issue as Open (in progress), as shown in Figure 3b, without further action required by the user. Sciit determines this state change because there is a commit on the feature branch dated newer than when the issue first appeared in a commit on the master branch.

As each sub-issue is completed, the developer deletes the comment describing the task in the appropriate module and makes a commit to the branch in the SCM. Eventually, all three tasks are completed and the author makes a final commit in which the original issue markdown file is deleted from the feature branch. Reviewing the status of the project in Sciit, we see that the issue’s status has been automatically changed to Open (In review), as shown in Figure 3c. Sciit assumes that since the issue no longer exists in the feature branch, the author is awaiting approval for the completed changes to be merged into master. Finally, the feature passes the team’s QA review process and the developer merges the branch into master, which causes the issue to be removed from master. The issue is now marked as Closed (resolved), as shown in Figure 3d. Alongside the change in status, Sciit also automatically adds a closed time to the issue.

At the end of the sprint, the whole team reviews progress on the project. The full revision history of an issue can be reviewed using the command line interface, as well as more detailed metadata, such as the location of the issue within the source file, as shown in Figure 4. For each change made to an issue, Sciit reports the identifier of the contributor who made the commit, the date of the commit, an itemised list of changes to the issue and the commit message summary. The set of commits that the issue was present in is also listed. The issue tracker also reports the full history of all branches and file paths that the issue resided in during its life-cycle. Supplementary metrics, such as the duration of the issue (the time between the first in progress commit and the issue being closed) are also reported.

To recap, a lifecycle for an example issue is illustrated in Figure 5. An issue is created in a new feature branch as a markdown document, or within a source code comment. Sciit reports this issue as ‘Proposed’. When the feature branch containing the issue is first merged to the master branch it is reported as ‘Accepted’. Following the first commit to the feature branch after the issue appearing in the master branch, the issue is reported as ‘In progress’. Deleting the issue content from the feature branch changes its status to ‘In review’. Finally, when the issue is removed from the master
Feature: Photo upload for claims

Scenario: Small JPEG Upload with Description

Given a claim
And a small JPEG
And a description
When I select the photograph
And I enter a description
And I click submit
Then the photograph is stored
And a database entry is created.

---

Figure 3: Making progress on an issue. The developer extends the feature file in the sub-issue branch (3a). Sciit detects that a commit in the feature branch means that work on the issue is now ahead of the status in master and so reports the feature as Open (In progress) (3b). Issues are automatically marked as in QA by removal from the respective feature branch (3c). Issues are marked as Closed (resolved) when removed from master due to merge (3d).
Figure 4: The full history of an issue stored within Git can be reviewed using the Sciit command line or web console.
branch, by (for example) merging the feature branch back to master, the issue status changes to ‘Closed (Resolved)’.

3. Implementation Details

Figure 6 illustrates the software components of Sciit, implemented on top of a peer-to-peer network of git repositories. The figure shows that Sciit comprises a library for managing a cache of issue snapshots extracted from a git repository; a command line user interface git extension; and a Gitlab integration that ensures consistency between a Gitlab issue database and a Sciit git issue repository. Notice that interactions between repositories that store Sciit issues is undertaken entirely by normal Git push/pull operations. The Sciit implementation works entirely on a local repository and requires the Git infrastructure and network for storage and communication.

3.1. Sciit Library

The core of Sciit is a python library implemented on top of git-python [7]. The library has two main functions: maintaining a cache of issue snapshots as commits are added to the repository; and reconstructing an issue history from the cache when interrogated by clients, such as the command line interface and Gitlab integration service.

The cache update mechanism is invoked on a repository following commits, merges or pulls. Sciit is notified of these events by hooks in the host git repository. Sciit first determines which commits in the repository have not yet been cached. The set of changed file paths for each of these is then extracted from the repository. Each file is then scanned for changes to issues. Sciit assumes that issues are contained within comment strings of the language indicated by the file’s extension. Issue snapshots in files with a .java extension are assumed to be contained within /*...*/ comments, for example. All issues are marked by an @issue tag. Any issue found within a file in the change file paths is assumed to have potentially changed. In addition, any issue that was in one of the file paths that changed, but is no longer present is assumed to have been deleted from the commit. All other issues are assumed to be unchanged in the commit. Each issue snapshot is then cached to an Sqlite database within the git repository. A snapshot in the cache records the details of the issue within the commit, the digest of the commit and the branches that the commit belongs to.
Figure 5: Lifecycle of an issue managed within Sciit.
A key feature of the Sciit library is that every commit is checked to ensure that an issue only occurs once within the source code base in the same commit. This check is applied by the post commit hook. If the hook discovers that the commit contains duplicate issues the commit is rejected and the repository is rolled back to its previous state.

Sciit uses the cache of snapshots taken from commits to build the history of issues. All snapshots are retrieved from the cache and grouped by issue id, allowing the entire history of an issue to be inspected. The current status of each issue for many fields, such as title, description and last modification can be extracted directly from the latest snapshot for an issue. More complex properties can be inferred from analysis of the entire snapshot history. For example, the set of participants in an issue can be inferred from the usernames of commit authors who made modifications to issues. The implementation time of closed issues can be calculated as the time difference between the first and last commits for an issue. The status of an issue can be inferred by its historical presence in different branches in the project repository. For example, an issue that is present in the head commit of a feature branch, but having never been present in the master branch is assumed to be open, but not accepted into the project.

3.2. Command Line Git Extension

Sciit incorporates an extension to the Git command line user interface that enables users to review and modify issues within the repository. Appendix C summarises the commands. Full details of options can be found in the project documentation. Issues can be presented in several different
views including a status summary of all issues in the repository; a tracker view providing detailed information about a single issue, including its full history; and a log view providing a summary of any changes to an issue in a single commit.

Several commands have also been implemented to combine workflows steps that would otherwise have to be executed separately using git. For example, the `new` command creates a new branch, creates a new markdown file for an issue, performs a commit and then returns the user to the starting branch. This command also supports the creation of sub issues by enabling further issue branching from feature branches. Similarly, the `close` command locates and removes comments in the head commit of every branch an issue is present in, causing its status to change to closed in Sciit. The command line user interface can also be used to initialise and, if necessary, rebuild the Sciit issue snapshot cache.

### 4. Gitlab Integration Service

Early on in the project it was recognised that some users would be initially uncomfortable with the transition to a new interface and architecture for issue tracking. Equally, many projects that employ distributed version control still maintain a single common team repository, often deployed on a service such as Gitlab. To accommodate these use cases, Sciit incorporates a Gitlab integration service, that is able to synchronise issue information between the issue database and Git repository for a project maintained on Gitlab. A demonstration instance of the integration service has been deployed on the Python Anywhere service.³

The integration service makes use of two features of the Gitlab platform: the REST API and Webhooks. Gitlab enables projects to be configured with Webhook end points so that remote services can be notified of project events. The integration service makes use of this to be notified whenever changes are pushed to a project repository, or changes are made to an issue. When this occurs, the integration service uses the Gitlab API to make changes to the project hosted on Gitlab.

When an issue is changed via the Gitlab Web user interface, the integration service applies these changes to a local mirror repository on the

³[https://sciit.pythonanywhere.com/configure](https://sciit.pythonanywhere.com/configure)
integration service. The changes in the mirrored repository are then pushed
to the Gitlab repository for the project.

If a change is pushed to the Gitlab repository that contains changes to
an issue, the Gitlab API is then used to update the issue in the Gitlab
issue database. Changes made by the integration service are done by a
dedicated user account with permissions on the project being managed by
the integration service. The account is setup by the user during configuration
of the project with the integration service. Sciit can use identifier of the Sciit
user account to filter reports of the changes that arrive via the Webhooks
and avoid a infinite cycle of Issue and Push hook notifications.

5. Comparison with Other Distributed Issue Trackers

We are not aware of any work in the academic literature directly concern-
ing distributed issue tracking architectures, or their evaluation in practice.
However, several tools have come close to the vision represented by Sciit, in
terms of a close alignment of issue tracking and distributed source control
management. In this section, we review other distributed issue trackers (a
summary is shown in Appendix A) and demonstrate that the key advances
of Sciit are the treatment of issues as first class control items, enabling in-
fERENCE of issue state directly from the repository where work is focused; and
second that multiple user interfaces are provided to support multiple user
roles.

As can be seen from the table, other approaches can typically be cate-
gorised either as extensions to existing distributed SCM tools, such as Sciit
itself and Bugs Everywhere, or distributed SCM tools with integrated issue
trackers, such as Fossil and Veracity. In the former case, extensions may be
for a specific SCM, such as Git or Mercurial, or may support multiple ‘back
end’ data stores for the issue tracking data.

Issue data storage is a key design decision for distributed issue track-
ers. Two particular approaches are identified in the review of tools. Some
tools, such as TicGit and Git-bug store issues within the underlying SCM’s
database, and place the issue data within a specific branch or ref. The im-
plication of this approach is that there is only a single variant of an issue at
any one time within the issue repository, although changes are still tracked.
Although this simplifies issue management, the relationship between issue
state and work progress is loosened, since different aspects of an issue may
be addressed in several different branches.
Other tools such as Bug and b store issues in a plain text format directly within an SCM’s source tree, either using a JSON format for the file, or a directory hierarchy and file name conventions to indicate structure. This enables issues to exist in multiple branches at once. In addition, the bug project goes further, by allowing issues to be stored in multiple ‘issue’ directories throughout a project’s source tree. This enables issues to be stored closer to where the work done to resolve them will be undertaken.

Sciit goes beyond these approaches, allowing issues to be treated just the same as any other artefact in the source control system, rather than providing a separate data storage format for them. Issues can be created anywhere within a text file in the SCM repository, allowing a closer alignment between Sciit issues and the work to be undertaken to resolve them.

A second concern for distributed issue tracking is the presentation of issue data held in storage. All the tools reviewed incorporated a command line user interface (CLI), reflecting the focus on developer usage for distributed tracking, rather than other stakeholders. The CLI for all tools allowed issue information to be both viewed and modified. A key benefit of the close alignment to source and trees and branches within Sciit is that many of the properties of an issue, such as status, component and participants to be inferred directly from the state of the issue in the repository, rather than requiring these to be updated manually. Unlike other approaches, Sciit relies on Git for primary data storage, but also maintains a cached representation of issues drawn from Git commits in order to present information efficiently on the user interface.

However, Bertram et al. [8] notes that different users of an issue tracker have different needs and thus demand different views on the same data. Addressing this, several tools incorporate web based views of issue information, potentially easing the migration from other issue trackers. Several tools, such as Git-bug, support import of data from web based issue trackers, such as Github and Gitlab. However, none of the tools reviewed provide for synchronisation between a centralised and distributed issue tracker. In our approach to Sciit, we have developed three different user interfaces (command line, web and Gitlab integration) to accommodate the needs of developers, business analysts and project managers. However, further work is needed to understand the different needs of these different roles.
6. Conclusions

Issue trackers play a pivotal role in the coordination of a software project. Nonetheless, they are a significant source of redundant information because much of the information can already be inferred from the SCM, the primary representation of work for a software project. This paper presents Sciit, a distributed issue tracker, in which issues are treated as first class control items within a SCM. Although a particular software workflow and issue type were used to illustrate Sciit, the purpose is to demonstrate that the status of any software task and workflow is best elicited from the focus of that work, the source control repository, rather than secondary systems, such as issue trackers.

We plan further integrations for Sciit with popular software engineering tools, including integrated development environments and web based issue trackers to continue to develop Sciit. Many new features have also been proposed during discussions of Sciit, or by reviewers. For example:

- Allow attachments to be associated with an issue, such as screenshot images and error logs. Although this can currently be done manually, by embedding references to images in markdown descriptions, they do not currently appear as attachments in command line or Gitlab views. Particularly large attachments could make use of Git’s Large File Storage facility.

- Inference of more sophisticated types of metadata based on the state of artefacts in the SCM. Examples include exploiting change set characteristics (such as files and/or lines changed) to estimate costs for tickets.

- Extend support for single line comment issues, similar to conventional ‘TODO’ statements. This would allow the unification of issue tracking and ‘micro’ tasks within a single reporting system.

- Provide better support for workflow management and regulation. This could include, for example, better support for merging issues when two branches are merged. In addition, rules could be implemented to prevent developers from violating a workflow involving an issue. For example, it may be undesirable to delete an issue from master without having first removed it from a feature branch. A ‘force’ option for overriding the rules could also be provided.
We also plan a more extensive study of the benefits of using distributed issue tracking within real software projects. We plan to adopt a twin-track strategy, identifying open source projects that utilise distributed issue tracking and studying them empirically; and developing case studies of teams using Scikit or other distributed issue trackers. The dearth of academic literature in this area is a significant gap. Case study research of such experiences is necessary to understand how to best adapt existing software workflows to distributed issue tracking. Studies of this form also provide insights as to potential analytics for a project that can be derived from issue tracking data stored in Scikit. We anticipate that longer term studies will generate significant insights for the development of successful distributed issue tracking systems.

References


management and issue tracking architectures, in: 2019 IEEE Interna-
tional Conference on Software Maintenance and Evolution, ICSME 2019,
Cleveland, OH, USA, September 29 - October 4, 2019, IEEE, 2019, pp.
402–405.

gitpython.readthedocs.io/en/stable/.

[8] D. Bertram, A. Voida, S. Greenberg, R. Walker, Communication, collabor-
ation, and bugs: the social nature of issue tracking in small, collocated
teams, in: K. I. Quinn, C. Gutwin, J. C. Tang (Eds.), Proceedings of
the 2010 ACM Conference on Computer Supported Cooperative Work,
CSCW 2010, Savannah, Georgia, USA, February 6-10, 2010, ACM, 2010,
pp. 291–300.
## Appendix A. Comparison of Distributed Issue Trackers

<table>
<thead>
<tr>
<th>Name</th>
<th>SCM</th>
<th>Storage</th>
<th>Multiple branches</th>
<th>Web</th>
</tr>
</thead>
<tbody>
<tr>
<td>Git-issue(^a)</td>
<td>Git</td>
<td>Plain text files in dedicated directory</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>TicGit(^b)</td>
<td>Git</td>
<td>Dedicated branch</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Git-bug(^c)</td>
<td>Git</td>
<td>Dedicated branch</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Git-dit(^d)</td>
<td>Git</td>
<td>Dedicated branch</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Bug(^e)</td>
<td>Git, Hg</td>
<td>Plain text files in dedicated directory</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Issue(^f)</td>
<td>None</td>
<td>JSON files in dedicated directory</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bugs Everywhere(^g)</td>
<td>Multiple</td>
<td>SCM database objects</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>b(^h)</td>
<td>Hg</td>
<td>Plain text files in dedicated directory</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Veracity(^i)</td>
<td>Veracity</td>
<td>SCM database objects</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fossil(^j)</td>
<td>Fossil</td>
<td>SCM database objects</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Sciit(^k)</td>
<td>Git</td>
<td>Source code comments</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\(^a\)https://github.com/dspinellis/git-issue  
\(^b\)https://github.com/manveru/ticgit  
\(^c\)https://github.com/MichaelMure/git-bug  
\(^d\)https://github.com/neithernut/git-dit  
\(^e\)https://github.com/driusan/bug  
\(^f\)https://github.com/marekjm/issue  
\(^g\)http://www.bugseverywhere.org  
\(^h\)https://foss.heptapod.net/mercurial/b  
\(^i\)http://veracity-scm.com  
\(^j\)http://www.fossil-scm.org  
\(^k\)https://gitlab.com/sciit/sciit

Table A.2: Summary of other distributed issue tracking implementations.
Appendix B. Supported Files and Styles

Since issues are embedded in block comments, there are different styles of block comments and files that support those types.

C Style Languages

Supported file extensions:

- C/C++ files .c, .cpp, .cxx, .h, .hpp, .hxx
- C# files .cs
- Java files .java
- PHP files .php
- CSS files .css
- JavaScript files .js
- SQL files .sql
- Scala files .scala
- Swift files .swift
- Go files .go
- Kotlin files .kt, .kts

/*
 * @issue Eg: The title of your issue
 * @description:
 * A description of an issue as you want it to be even with ‘markdown’ supported
 * @assignees to nystrome, kevin, daniels
 * @due_date 12 oct 2018
 * @label in-development
 * @weight 4
 * @priority high
 */
**HTML Style**

Supported file extensions:

- HTML files .htm, .html, .xhtml

<!--

@issue Eg: The title of your issue
@description:
   A description of an issue as you
   want it to be even with ‘’markdown‘’ supported
@assignees nystrome, kevin, daniels
@due_date 12 oct 2018
@label in-development
@weight 4
@priority high
--> 

**Python**

Supports .py files

```python
###
@issue Eg: The title of your issue
@description:
   A description of an issue as you
   want it to be even with ‘’markdown‘’ supported
@assignees nystrome, kevin, daniels
@due_date 12 oct 2018
@label in-development
@weight 4
@priority high
```

**MATLAB**

Supports .m files

```matlab
%%
@issue Eg: The title of your issue
@description:
   A description of an issue as you
   want it to be even with ‘’markdown‘’ supported
```
@assignees nystrome, kevin, daniels
@due_date 12 oct 2018
@label in-development
@weight 4
@priority high
%

Haskell
Supports .hs files
{-
    @issue Eg: The title of your issue
    @description:
        A description of an issue as you
        want it to be even with ‘‘markdown‘‘ supported
    @assignees nystrome, kevin, daniels
    @due_date 12 oct 2018
    @label in-development
    @weight 4
    @priority high
-}

Markdown
Supports .md files
---
@issue Eg: The title of your issue
@description:
    A description of an issue as you
    want it to be even with ‘‘markdown‘‘ supported
@assignees nystrome, kevin, daniels
@due_date 12 oct 2018
@label in-development
@weight 4
@priority high
---
Others

Supported file extensions:

- Ruby files .rb
- BDD feature files .feature
- YAML files .yml, .yaml
- Plain text files

```#
@issue Eg: The title of your issue
@description: A description of an issue as you want it to be even with ‘markdown’ supported
@assignees nystrome, kevin, daniels
@due_date 12 oct 2018
@label in-development
@weight 4
@priority high
```
Appendix C. Summary of Sciit Commands

High level usage:

```
git -sciit | git sciit | git-sciit.exe <command> <options>
```

**Common options**

Options below are common to two or more commands.

- `[--help | -h]` show the help message and exit
- ` revision` the revision path to use to generate the issue log e.g. ‘all’ for all commits or ‘master’ for all commit on master branch or ‘HEAD^2’ from the last two commits on current branch. See `git rev-list` options for more path options
- `[--full | -f]` view the full information for issues including, description, commit activity, multiple file paths, open in, and found in branches
- `[--normal | -n]` default: view summary issue information
- `[--all | -a]` show all the issues currently tracked and their status
- `[--closed | -c]` show only issues that are closed
- `[--open | -o]` default: show only issues that are open

**Init**

```
git sciit init [-r | -s]
```

creates an empty repository or builds from past commits

**Options:**

- `[--reset | -r]` resets the issue repo and rebuild from past commits
- `[--synchronize | -s]` synchronizes repository with remotes before initialisation
Status

```
git sciit status [-f | -n] [-a | -c | -o] [revision]
```
shows how many issues are open and how many are closed on all branches

Log

```
git sciit log [revision]
```
shows a log that is similar to the git log but shows open issues

Issue

```
git sciit [-f | -n ] issue_id [revision]
```
shows information about the issue with the given id

New

```
git sciit new [-p | -a]
```
creates a new issue in the project backlog on a branch specified by the
issue id

Prompts user for:

- Issue title
- Issue id or default slug of issue title
- File path for markdown file or default to backlog folder
- A description of the issue
- A commit message for creating the issue

Options:

```
[--push | -p] pushes the newly created issue branch to the origin
[--accept | -a] accepts the newly created issue branch by merging it
to master locally
```
Close (Experimental)

```bash
git sciit close issue_id
```
removes an issue from the master branch

The effect within Sciit is to change the issue status to Closed.

Web

```bash
git sciit web
```
launches a local web interface for the sciit issue tracker

Tracker

```bash
git sciit tracker [-a | -o | -c ] [-f | -n ] [revision]
```
shows a summary of issues and their status

Gitlab

Start

```bash
git sciit gitlab start
```
launches the gitlab webservice that integrates Gitlab issues with sciit

Used primarily for testing the Gitlab integration service. The command must be run in the parent directory of `gitlab-sites`.

Reset

```bash
git sciit gitlab reset project_url sites_local_path
```
resets the issue tracker database and rebuild from past commits

Arguments:

- `project_url` the URL of the project to be managed by Gitlab
- `sites_local_path` the path to the local gitlab sites mirror directory

Set credentials

```bash
git sciit gitlab set_credentials project_url gitlab_username \  
   web_hook_secret_token api_token sites_local_path
```
sets a gitlab username, web hook token and API token for a Gitlab project to be used by the sciit gitlab service
Primarily used for debugging purposes by integration service Admins.
The configuration process is largely automated by the Gitlab integration
service configure page.

Arguments:

- **project_url** the URL of the project to be managed by Gitlab

- **gitlab_username** the Gitlab account username that will manage the
  Gitlab issue tracker and repository synchronisation. The account should
  *not* be that of an actual developer on the project.

- **web_hook_secret_token** token set on the Gitlab repository to authen-
ticate incoming Webhook events.

- **api_token** token set on the Gitlab user account to provide access to the
  Gitlab API.

- **sites_local_path** The path to the local gitlab sites mirror directory
### Required Metadata

#### Current executable software version

<table>
<thead>
<tr>
<th>Nr.</th>
<th>(executable) Software metadata description</th>
<th>Please fill in this column</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Current software version</td>
<td>2.1.2</td>
</tr>
<tr>
<td>S2</td>
<td>Permanent link to executables of this version</td>
<td><a href="https://gitlab.com/sciit/sciit/-/releases#v2.1.1">https://gitlab.com/sciit/sciit/-/releases#v2.1.1</a></td>
</tr>
<tr>
<td>S3</td>
<td>Legal Software License</td>
<td>MIT Licence</td>
</tr>
<tr>
<td>S4</td>
<td>Computing platform/Operating System</td>
<td>OS X, Microsoft Windows, Unix-like</td>
</tr>
<tr>
<td>S5</td>
<td>Installation requirements &amp; dependencies</td>
<td>See <a href="https://sciit.gitlab.io/sciit/">INSTALL.md</a> and <a href="https://sciit.gitlab.io/sciit/">requirements.txt</a></td>
</tr>
<tr>
<td>S6</td>
<td>If available, link to user manual - if formally published include a reference to the publication in the reference list</td>
<td><a href="https://sciit.gitlab.io/sciit/">https://sciit.gitlab.io/sciit/</a></td>
</tr>
<tr>
<td>S7</td>
<td>Support email for questions</td>
<td><a href="mailto:timothy.storer@glasgow.ac.uk">timothy.storer@glasgow.ac.uk</a></td>
</tr>
</tbody>
</table>

Table C.3: Software metadata (optional)

#### Current code version
<table>
<thead>
<tr>
<th>Nr.</th>
<th>Code metadata description</th>
<th>Please fill in this column</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Current code version</td>
<td>2.1.2</td>
</tr>
<tr>
<td>C2</td>
<td>Permanent link to code/repository used of this code version</td>
<td><a href="https://gitlab.com/sciit/sciit/-/releases#v2.1.1">https://gitlab.com/sciit/sciit/-/releases#v2.1.1</a></td>
</tr>
<tr>
<td>C3</td>
<td>Legal Code License</td>
<td>MIT Licence</td>
</tr>
<tr>
<td>C4</td>
<td>Code versioning system used</td>
<td>git</td>
</tr>
<tr>
<td>C5</td>
<td>Software code languages, tools, and services used</td>
<td>Python 3.7 or later, Sqlite 3.8.2 or later, Git</td>
</tr>
<tr>
<td>C6</td>
<td>Compilation requirements, operating environments &amp; dependencies</td>
<td>See INSTALL.md and requirements.txt</td>
</tr>
<tr>
<td>C7</td>
<td>If available Link to developer documentation/manual</td>
<td><a href="http://sciit.gitlab.io/sciit/">http://sciit.gitlab.io/sciit/</a></td>
</tr>
<tr>
<td>C8</td>
<td>Support email for questions</td>
<td><a href="mailto:timothy.storer@glasgow.ac.uk">timothy.storer@glasgow.ac.uk</a></td>
</tr>
</tbody>
</table>

Table C.4: Code metadata (mandatory)
**Nystrom Edwards**: Conceptualization, Software, Writing, Visualization.

**Dhitiwat Jongsuebchoke**: Conceptualization, Software, Writing, Visualization.

**Tim Storer**: Conceptualization, Software, Writing, Visualization, Supervision, Project Administration.