Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis

Eventual Consistent Issue Synchronisation between Gropius and Traditional Issue Management Systems

Christian Kurz

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Sandro Speth, M.Sc.
Commenced:	June 21, 2023
Completed:	December 21, 2023

Abstract

Context. In the development of component-based systems, often issues affect multiple components or must reference issues of a different component, resulting in the Cross-Component Issue Management System, Gropius being invented. Gropius needs to synchronize issues with traditional per-component Issue Management Systems (IMSs), e.g., GitHub or Jira, to have relations between issues on different IMS.
Problem. The existing system can currently communicate exclusively with GitHub supporting only a limited subset of the required features. It is susceptible to duplicating issues and timeline items potentially infinite times due to missing identifying meta information.
Objective. We provide a redesign of the Gropius Sync, designed to be modular and able to host multiple methods of duplicate detection.
Method. We implement the new Sync Framework, adapters for GitHub and Jira as representatives for traditional IMSs and compare different methods for duplicate prevention.
Result. We succeeded with our plans to provide a new Sync that can both sync to GitHub and Jira and thus demonstrate the feasibility of the sync. We successfully detect and prevent most previously found issue duplication loops.
Conclusion. The new Gropius Sync allows synchronization between multiple IMS Projects without runaway duplication of issues.

Kurzfassung

Kontext. In der Entwicklung von komponentenbasierten Systemen betreffen Issues häufig mehrere Komponenten oder müssen auf Issues anderer Komponenten referenzieren, weshalb das komponentenübergreifende Issue Management System Gropius entworfen wurde. Gropius muss Issues mit traditionellen, komponentenspezifischen Issue Management Systemen (IMSs), wie GitHub oder Jira, synchronisieren, um Verbindungen zwischen Issues auf verschiedenen IMS zu haben.
Problem. Das bisherige System kann ausschließlich mit GitHub kommunizieren und unterstützt nur eine limitierte Untermenge der benötigten Funktionen. Es ist anfällig für das potenziell unendliche duplizieren von Issues und Timeline Items (Zeitstrahlobjekten) aufgrund von fehlenden Metainformationen.
Ziel. Wir beabsichtigen eine Neuentwicklung des Gropius Sync, welche modular ist und Duplikatsverhinderhung unterstützt.
Methode. Wir implementieren den neuen Sync und die Unterstützung für Adapter zu GitHub und Jira als Repräsentanten traditioneller Issue Management Systeme und vergleichen verschiedene Methoden der Duplikatsverhinderung.
Resultat. Wir haben erfolgreich den neuen Sync erstellt, welcher sowohl GitHub als auch Jira ansprechen kann und damit die Möglichkeit des Syncs demonstriert. Wir können erfolgreich die meisten bisher gefunden Duplikationsschleifen detektieren und verhindern.
Schlussfolgerung. Der neue Gropius Sync ermöglicht die Synchronisation zwischen mehreren Projekten ohne endlose Duplizierung von Issues.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 1

IDE Integrated development environment. 6

IMS Issue Management System. vii

JSON JavaScript Object Notation. 7

NLP Natural language processing. 18

REST Representational state transfer. 22

SLO Service-Level Objectives. 5

TOSCA Topology and Orchestration Specification for Cloud Applications. 6

UI User interface. 31

UUID Universally Unique Identifier. 15

1 Introduction

Component-based systems can have issues that affect more than one component. But often each component uses a different IMS to track their issues. To work with this and manage issues that affect multiple components, the Cross-Component Issue Management System, Gropius has been designed. As not all stakeholders can be forced to switch to Gropius, it has been designed to sync with conventional IMS. This allows Gropius to transparently work with components that do not use Gropius as their main IMS and even allows stakeholders to access Gropius-managed projects via conventional IMS.

In the current Gropius codebase, a *Sync* is included which only supports syncing issues from and to *GitHub*[1]. Furthermore, as it is deeply integrated with the *GitHub* Application Programming Interface (API) a port to different IMS has been deemed not feasible, as it exploits the similarity of the models of Gropius and *GitHub*, which makes porting it to IMS with a different model basically a rewrite. It only supports a limited subset of the required features and can in many cases duplicate issues and timeline items that are not intended to be duplicated. This happens due to the conceptual loss of some meta information and can even lead to the continuous generation of duplicates.

We provide a redesign of the Gropius *Sync* which fixes these issues. First, we provide an IMS agnostic core framework and module set to allow easy building of new sync adapters. Next, we will implement two different adapters, one for *Jira*[2] and one for *GitHub* using this core framework. We only target eventual consistency, as the IMS usually have rate limits that limit how much the *Sync* can even interact with the IMS forcing it to drag out operations over multiple API limitation budgets, as well as other limitations given by the network topologies. We incorporate it into the design to be able to work with different methods of duplicate detection, as we have found no perfect way to recover the meta information lost when syncing to an IMS to prevent merging of different issues or timeline items or duplication of an issue or timeline item. We implement multiple alternative methods of duplicate detection and evaluate these, as well as the *Sync* itself. Our evaluation also shows if the *Sync* can work with real-world data and keep stable copies of projects ready for use in Gropius.

We will provide a framework for the rapid development of new sync adapters, two adapters, and multiple duplicate detection modules as well as an evaluation of the abilities of the *Sync* as well as the duplicate detectors. This results in a Gropius *Sync* that allows synchronization between multiple IMS Projects without runaway duplication of issues.

[1] https://github.com/

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

Thesis Structure

The full structure of the thesis is as follows:

Chapter 2 – Foundations and Related Work: Introduces Gropius as well as all other concepts and technologies needed to understand the thesis. Furthermore, we elaborate on related work regarding synchronization of issues between different IMS.

Chapter 3 – Concept: This chapter describes the concept of our *Sync* and the basics of issue replicators.

Chapter 4 – Architecture & Implementation: We detail the architecture and implementation of the software.

Chapter 5 – Evaluation: Here, we evaluate our framework using real-world data and multiple constructed test cases.

Chapter 6 – Conclusion We conclude our thesis and give an overview of our results and the remaining future work.

2 Foundations and Related Work

In this chapter, we introduce the foundations required for the remaining thesis in Section 2.1, and the related work with regards to the synchronization of issues in Section 2.2.

2.1 Foundations

This section describes the foundation needed to understand the thesis. First in Section 2.1.1, we explain what issues are and how IMSs track them. Section 2.1.2 then describes the component-based software architecture style. The combination of both is in Section 2.1.3 and how it creates the Gropius method can be found in Section 2.1.4 with special focus on the *Sync* part in Section 2.1.5.

2.1.1 Issue Management System

Issues are a typical way to manage features, bugs and quality of software in a structured, searchable and formal manner [CV85]. An issue usually consists of at least a title, a description, a state, like open or closed/resolved, and a list of comments [BJS+08]. All issues concerning a piece of software are then managed in one instance of IMSs. The IMS is the software that manages, saves and displays the issues, popular ones being *Jira* or integrated parts of the Version Control System,

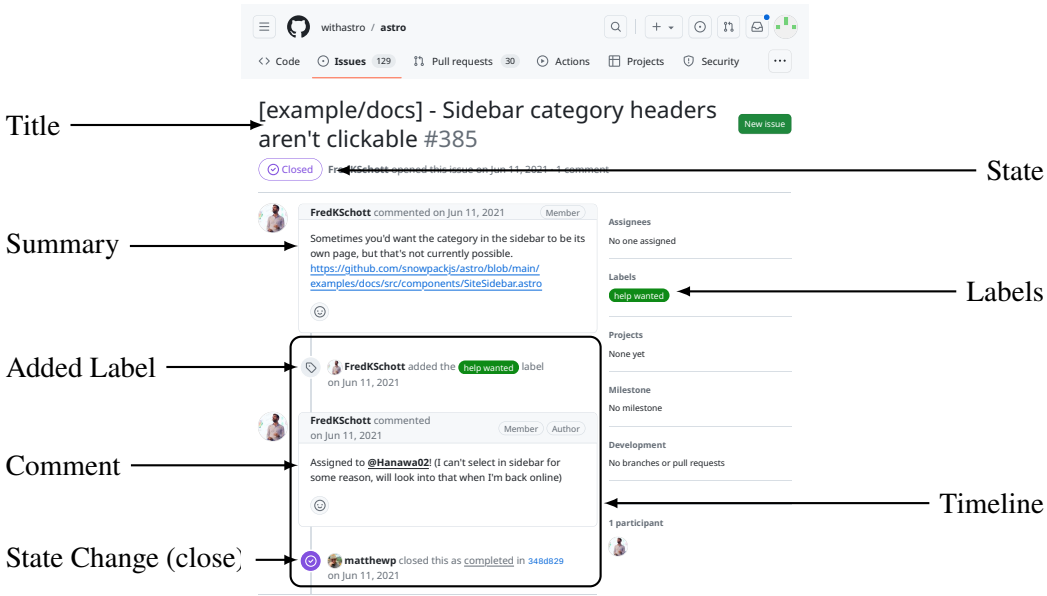


Figure 2.1: One Issue on the *GitHub* Issue Management System.

like for *GitHub* [AMR20]. The title of an issue is usually written by the original author and serves as a very short summary of the affected parts and a very rough indicator of which problem is being discussed. The title is followed by a description in which the author is supposed to describe enough of the context to understand why the issue exists and in case of bugs contains ideally the necessary information to find and debug the issue. While both the title and description usually have to be filled out by the original author of the issue, the following comments can then be commented on by people working on the project or can be written by the original author as a way to provide further information. This usually means that the list of comments provides a discussion between the stakeholders involved in an issue, be it developers, quality assurance or sometimes even the consumers of the software. Ideally, at some point, this leads to an issue being solved, which is the point where the root problem that has resulted in the issue being opened has been repaired or otherwise made irrelevant. This should then be reflected in the state of the issues, which is at least an indicator between open, which means that the issue is not yet unsolved and closed meaning the issue has been solved. Another important part of a single issue is the timeline, which logs all changes to the issue and is usually interleaved with the comments to provide context for the order and timing in which events related to the issue have happened.

Usually, these basic features are not enough and a few common additions have become popular. The most common additional features are labels, which are tags that can be attached to an issue and then be searched for. Common labels are “bug” or “feature” allowing users to specifically filter for only bugs or only feature requests. Another common feature is the mentioning of different issues which allows one to specify other issues inside of a comment, which then puts a link the the original issue on the timeline of the mentioned issue. This feature is usually only available inside of one IMS, which restricts the targets that can be linked using this feature. One screenshot of an example issue has been provided in Figure 2.1.

2.1.2 Component-Based Software

Large software cannot be a uniform substance anymore but instead consists of components [Crn03; SGM02]. These components are usually designed before implementation and are then combined to form the software. The components should have specified tasks and interfaces to interact with each other [Nyg18]. Aside from bringing structure into software, components also provide a way of splitting the software into parts which then can be worked on by multiple people or even multiple groups. Another feature of software components is the ability to share them over multiple projects, allowing to save time and thus development costs.

In recent years there has been a trend to even outsource the development of some components to other companies [MK11]. This results in being able to have more specialized teams working on the components and further fostering the interchangeability of the components.

If taken further and making each component into its own application the system architecture becomes microservice architecture, meaning that the system is composed of of many interacting standalone applications [RMBZ21; Thö15]. This greatly improves scalability on cloud platforms as it allows to just start of more instances running the saturated services thus improving overall performance. Another important improvement of the development of microservices is that a clear separation is enforced by the concept [TLPJ17]. Also as each team is concerned with one microservice they can more independently develop and test the chance of accidentally implementing features intended for

other parts of the system is reduced significantly. The same independence also allows teams to use their preferred tech stack to use previous knowledge instead of having to train in the use of different technologies.

Another feature of component-based software is the ability to just buy access to a running instance component instead of having to develop, host or maintain yourself [DW07]. This reuses not just the code, but even the servers and the maintainers and allows the sellers to scale the software as one instead of having to keep each instance continuously ready.

2.1.3 Issue Management in Component-Based Software

As the number of different teams or even companies involved in a project grows, the management of issues gets more complex [BVGW10]. Usually, every project gets its project on an IMS, which is often hosted independently and can even be a different software. This means that for example in case of a single bug or feature request in one component having a single, easy-identifiable IMS where it can be reported, sometimes the root cause lies in a different component tracking their issues in a different IMS [MNH15; SG05].

This forces developers to pursue different methods of making connections between issues. The most common way is to write a link to the other issue inside a comment. But this is tedious when backtracking through the chain of linked issues and does not provide any other features that linked issues should have. For example, closing the root issue does not notify the dependent issues and all information is scattered over multiple issues. This usually leads to a problem that if an issue is caused by a defect in a different project, linking to the original issue does not propagate the issue back. This often leads to many open issues which are waiting for different projects that wait for fixes that have already been incorporated and just not noticed.

Alternatives that have been observed in the wild are the use of third-party messengers, emails or even in-person meetings [SBB20; Spe19].

2.1.4 Gropius

To solve this problem of linking issues between components, Gropius has been developed by Speth et al. [SBB20; SBB21; SBK+23] and Speth [Spe19; Spe21]. Gropius is an IMS intended to link and manage issues between different *components*. In addition to regular cross-component issue management, software architects can use Gropius to iteratively and incrementally refine software architectures [SSFB22]. In such a use case, feature requests often affect multiple components and must be synchronized between these components' underlying IMSs. Furthermore, DevOps engineers can annotate Gropius architectures with information about Service-Level Objectives (SLO) to analyze the impact of an SLO violation across the entire architecture and report explanations of the violation cross-component issues via Gropius as validated by Speth et al. [SBSB23] via experiments on the T2-Project microservice reference architecture [SSB22].

Gropius combines *projects* and *components* into *trackables* which represent everything that can contain *issues*. *Projects* are intended to represent general projects and are intended to contain mainly *issues* related to the whole architecture and new *components* that may be developed in the future. *Components* are referenced by version from *projects* and should contain the *issues* that

concern this component of software in the final product. The issues themselves consist of a *title*, the current snapshot and the *timeline items* that led to this snapshot. Example *timeline items* are a *state change timeline item* which changes the state of the *issue*, for example from open to close, *assigned timeline item* which assigns a user to the issue and the *issue comment* for the normal comment containing markdown text. To show links between issues, *issue relations* can be used to directionally specify a relation type for two issues. Issues can affect other parts in Gropius, for example, *projects* or *components*. Based on a previous version of Gropius, Neumann [Neu20] and Speth et al. [SKBB21] implemented a concept for an Integrated development environment (IDE) integration of Gropius allowing cross-component issue management in a component-specific view directly in the developer's IDE.

Gropius is an IMS specialized for tracking issues of multiple components, linking between issues and issues that affect multiple components. Contrary to traditional IMS, the issue cannot only lie on a single component, but on multiple components, on interfaces that connect components and the project itself. This allows issues to be transferred to the project responsible for the root cause without removing it from the initial reporter, removing all of the hassle and mistakes that would stem from having multiple issues for one problem.

Using templates, as specified later, Gropius has an extensible model that can represent or mimic all necessary features of other IMS. Behaving as a superset of other IMS allows Gropius to represent and work with data from external IMS, once they are imported using the *Sync* described in Section 2.1.5.

Snapshot and Timeline

Each issue has a timeline and a current state, which we call a snapshot. We define the snapshot of an issue as the current properties of an issue at one point in time. In the case of Gropius this includes the *title*, the current list of labels and if the issue is open or closed. The timeline consists of timeline items, which represent actions with a corresponding time that changes the snapshot. These actions usually describe snapshot changes in the issue. For example, if the *title* of an issue is changed, a corresponding *title changed event* is placed into the timeline. This allows the current snapshot of an issue to be reconstructed using *timeline items* and any previous valid snapshot. For this, the *timeline items* are designed to always specify the new value of their property. Recreating the current snapshot can be done by taking any previous snapshot and applying all timeline items to it. As each *timeline item* contains the new value of their property, this results in the current snapshot after all changed properties have been replaced. If a *timeline item* which is older than the starting snapshot is applied, this may result in an intermediate snapshot that has never happened, but after applying all other *timeline items*, we still arrive at the current snapshot nonetheless. The original snapshot is only necessary as a few values like the initial title not being saved in timeline items, so unless the title has been changed it does not appear in the timeline.

Templates

For flexibility Gropius uses a templating system. Our templates are similar to Topology and Orchestration Specification for Cloud Applications (TOSCA) *node types*, which define properties for TOSCA *nodes* in a typed way [BBLS12]. This means that many nodes, for example, issues

and IMS have a template associated. This template provides type-safe fields on the node. In the case of *issues*, a field could be used to specify the amount of story points a given feature has been assigned. As there is usually no template restriction, it is possible to tailor the fields of an issue as close to the intended use-case as wanted. To keep order and safety, each field is specified using a JavaScript Object Notation (JSON) schema. JSON schema specifies the type or schema a JSON object can have, meaning it restricts the content of the object to the specified format, forcing the objects to follow a given type descriptions [PRS+16]. This can be used for as simple tasks as restricting a templated field to number values and can be as complex as needed for any other possible data. The *Sync* only interacts very specifically with templated fields from nodes, which leaves only the following list of nodes as having templated fields that are relevant for the *Sync*: Issue, IMS, IMSIssue and, IMSProject.

Login Service

The Login Service takes care of the whole user management, including logging the user in and connecting accounts from IMSs to Gropius. The *Sync* is then able to query API tokens for a given user.

2.1.5 Gropius *Sync*

As mentioned before, some components may be acquired from external companies, resulting in more complex support than in-house developed components. Assuming these components are publicly using an IMS, it is necessary to somehow integrate their IMS as forcing them to use another instance of Gropius is not viable [SBB20; Spe19]. This is where the *Sync* comes in: It mirrors the contents of an IMS into Gropius and writes back changes. This also allows syncing a repo from one IMS to another, for example keeping a *GitHub* issue tracker synchronized with a *Jira* issue tracker.

This allows the full usage of Gropius features, like cross-referencing, issue annotating and more on non-gropius repositories. When writing the newly created data back to the original IMS, Gropius will use either a shared fake user or if possible the API access of a user who has connected their account to Gropius. Gropius can also pretend to be a user if the repository is not public or does not support access specific for this Gropius instance. If all users writing have connected accounts on the original IMS with their accounts on Gropius, this allows Gropius to be completely invisible to the original IMS.

The *Sync* should attempt to blend in as much as possible, to allow adding existing and public repositories without damage. We consider damage in this context to be any unwanted change to existing issues, for example, mass changes that tag the title of every issue with a Gropius generated ID.

2.2 Related work

This section describes the related work of this thesis. In Section 2.2.1, we describe our Literature Research Methodology. Section 2.2.2 discusses related research work and Section 2.2.3 describes related industrial efforts to synchronize issues across the same or different issue management systems.

2.2.1 Literature Research Methodology

Literature has mainly been collected by searching in Google Scholar[1] for the terms "issue", "synchronization" and "software development". For all terms, synonyms have also been tried, these being "work item" instead of "issue" as they are similar in our use case and "transfer" instead of "synchronization" for more unidirectional issue synchronization.

2.2.2 Literature Research Results

The thought of converting issues between different IMSs has been around for a long time. An early example would be the criticism by Jakobsen et al. [JFC+09]. While they do not work with issues specifically and instead use *Work Items* they seem comparable and can be taken as interchangeable in our use case. At some point they seemed to have a similar problem, being able to export their Work Items from Team Foundation Server to Excel but not back. Their resulting decision was to create a platform and force all teams to use it to plan their work and did not try to transparently keep the different IMS synchronized.

There also have been previous attempts at cross-platform issue synchronization [La17]. While not technically an IMS, Polarion handles the whole Application Lifecycle Management, but can also manage Work Items, which are in our case similar enough to issues. In this example, the Polarion Connector family of Polarion extensions can each sync a different Application Lifecycle Manager, which is their equivalent to IMS, to Polarion and back[2]. This is very similar to our *Sync* as both our usage is similar. Especially notable is the configurability to specify details of the transformation during synchronization and even limit the sync directions or even filter which items should be synchronized.

2.2.3 Industrial

To manage issues over multiple components, different IMS communities recommend different solutions: For Redmine users, if all components use the same Redmine project, the forums recommend using a single overarching task with subtasks for each component it affects but do not offer any useful methods if, as usual, the projects are on different Redmine projects[3]. *Jira*

[1] scholar.google.com

[2] https://extensions.polarion.com/extensions/190-polarion-connector-for-atlassian-jira

[3] https://www.redmine.org/boards/1/topics/21939

recommends the same[4], but also offers a plugin for if all component projects are *Jira* based to use the *Structure*[5] plugin to combine multiple projects into a spreadsheet, but as this requires owner permissions to all projects and forces them to be on the same *Jira* instance it also has limited practical usage [SBB20]. Other *Jira* plugins[6] offer more synchronization services, sometimes even over multiple projects to *Jira*, but do not support different IMS. For *GitHub*, linking between issues of different projects can either be done using *ZenHub*[7] or in a limited fashion integrated in the form of special markers in comments that leave a backlinking timeline item on the link target.

In recent years, new IMS have been conceived which provide a *Sync* to allow for a soft migration by using both IMS at the same time, like for example for Plane[8] or even for Github to and from *Jira*[9].

Plane is a new IMS that is intended to manage whole projects instead of just components. Contrary to Gropius, while it allows a *workspace* to consist out of multiple *projects*, that are similar to our components, it does not seem to support better linking between issues or other Gropius core features. The *plane GitHub* sync supports keeping a single *plane* project in sync with a *GitHub* repo. We did not find any signs of dereplication to combat duplicate issues caused by more complex topologies.

The mentioned *Jira-GitHub* sync is part of *Unito*[10]. It presents itself as a platform allowing a user to create custom rules of what is synced where. Using it with a *GitHub* and a *Jira* module allows bidirectional sync between *GitHub* and *Jira*. While it allows *Sync* between more than two projects, we think the docs imply that it requires all parts to be synced by a single *Unito* instance being the only *Sync* between the projects.

All syncs we found map between two IMS with hard rules specifying what to sync and do not support a single repository involving more complex topologies than a single chain by syncing from a synced repo.

[4] https://community.atlassian.com/t5/Jira-Software-questions/Share-one-issue-quot-ticket-quot-across-multiple-projects-and/qaq-p/407534

[5] https://community.atlassian.com/t5/Jira-questions/How-do-others-work-with-issues-affecting-multiple-projects/qaq-p/399950

[6] https://marketplace.atlassian.com/apps/1211709/multi-project-picker?hosting=server&tab=overview, https://www.k15t.de/software/backbone-issue-sync-for-jira

[7] https://www.zenhub.com/

[8] https://docs.plane.so/integrations/github

[9] https://unito.io/integrations/github-jira/

[10] https://unito.io

3 Concept

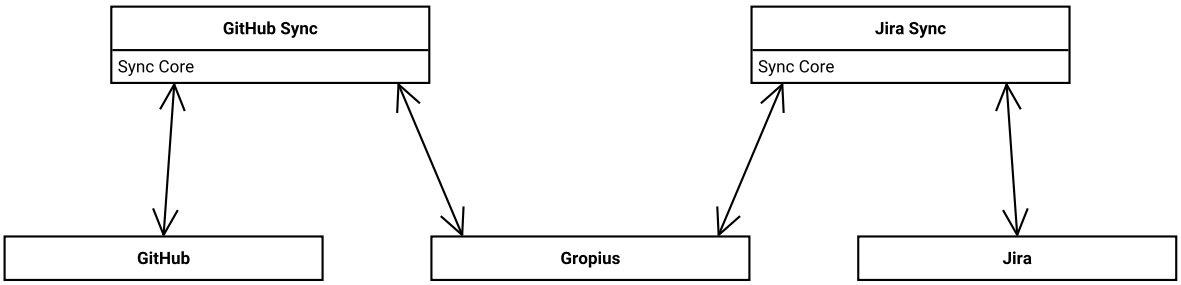


Figure 3.1: Example *Sync* Topology.

We plan to write a common core part for the *Sync* and two sync adapters that synchronize issues between Gropius and IMSs, as seen in Figure 3.1. First, we detail the concept of the *Sync* and the high-level summary of what an issue replicator is in Section 3.1. The sync adapters should mainly contain code specific to their IMS while the common core contains all relevant building blocks, as described in Section 3.2. Our previous simple synchronization algorithm tends to duplicate issues and timeline items, which we further investigate in Section 3.3.

3.1 Overview of the Concept

The cross-component issue management system, Gropius is intended to manage issues for multiple components. As existing third-party components already use their own technology stack and issue management system, we cannot expect them to use a joint Gropius instance instead, which we solve by using the Gropius *Sync* to import and keep synchronized with external IMSs. The *Sync* keeps the issues on a project or component in Gropius as similar as possible to a counterpart on an IMS. We do this by synchronizing issues and the timeline items the issues consist of. As we can assume issues and timeline items continue existing after they have been observed once, we do this by adding issues and timeline items wherever they are missing, ensuring an eventual consistent state on all participants. This allows Gropius not only to keep one component or project synchronized with an external IMS, but also keep two external IMS synchronized by using Gropius as intermediate, as demonstrated in Figure 3.2. We currently target *GitHub* and *Jira* as our first two adapters.

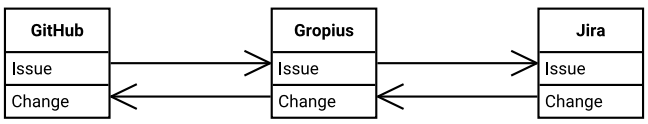


Figure 3.2: Common *Sync* example between multiple IMSs and Gropius.

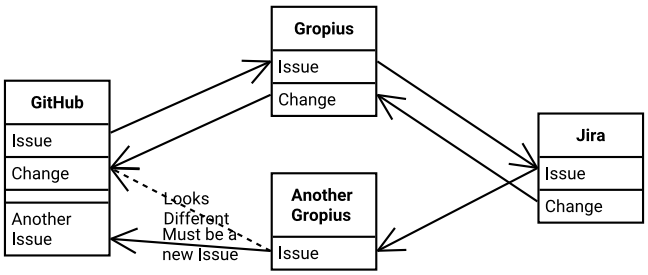


Figure 3.3: Abstract example showing the duplication of an issue.

One big challenge we have to face when it comes to synchronizing the same issue between multiple components' IMS projects is identifying the issues of the different IMS projects as the same. While we are able to retrieve the IMS projects' IDs for the issues and store them within Gropius, this is no longer feasible when multiple Gropius instances come into play. This might happen especially for open source components when two different companies use their private Gropius instances but include the same components in their projects. As a result, each instance manages the issue IDs themselves. If now one Gropius instance synchronizes a new issue to the other component, the other Gropius does neither store nor have access to the IDs of the issues, thus, missing the equality and synchronizing the newly synchronized issue back to the original component where it will again be synchronized by the first Gropius instance and so on. We call this behavior *issue replicators*. Unfortunately, managing the IDs of the issues shared over all Gropius instances is not feasible, e.g., due to a company's data privacy regulations. Hence, we require a solution to prevent or at least stop such issue replications early. In general, we developed one invasive and five non-invasive methods to avoid issue replications for issues in total and issue comments only by identifying issue equality without sharing the issue IDs with other Gropius instances.

3.2 *Sync*

The *Sync* should keep the Gropius and the target IMS consistent.

To support the querying of as many IMS as possible, the *Sync* should only use available APIs of the target IMS, to allow adding any IMS required by developers. We decided to accept the limitations of having to adhere to rate limits, multiple user accounts and missing out on multiple deeper features, over alternative options. The features we are thus unable to implement usually concern changing histories, meaning that all items will display the timestamp of the *Sync* that synced them instead of the original time as displayed in Gropius. This can lead to blocks of timeline items that all bear the identical date of a single sync cycle if multiple Timeline Items have been queued up inside the *Sync*. The main alternative would have been to access the database directly, which would limit us to only self-hosted IMS and introduce versioning incompatibilities and security risks. As our first target is *GitHub*, which is usually used as a service in the cloud, the choice was necessarily made to restrict ourselves to public APIs only.

To support rate limits the *Sync* has to be able to send the requests for data by spoofing as multiple user accounts. For this, the *Sync* has to figure out which account to get data from but can be balanced over multiple user accounts.

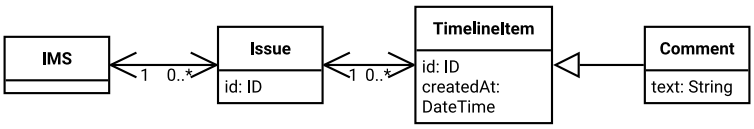


Figure 3.4: Minimum viable model.

The *Sync* requires a preset bot user, preferably a common user to fall back on in case a specific user is not available, some for fetching data and writing user-independent data. This role does not have to be fulfilled by a separate user and can be taken on by any account, we refer to it as a bot user because it is used for the tasks one would usually expect from a maintenance bot.

If a user has a linked account that is viable, the *Sync* should use it to simulate the action taken by the user indistinguishably. For example, if a user has an account linked and comments on an issue, we will create the comment with this account resulting in the user showing up on the target IMS. If the user has no linked account, we have to fall back to the bot user. It is important that, once a comment has been worked on by one user, we should be able to restrict all further actions to the same user, as many IMSs do not necessarily allow us to edit comments by different users. From the login service[1] the *Sync* can request for a user an account list and the respective login tokens.

As we want to be able to eventually support as many IMS as possible, we decided to force minimal restrictions on the target IMS. Conceptually, to be able to be interfaced with the *Sync*, we require on the remote side an equivalent of issues with a unique identifier and a title. Each of these *issues* must consist of *timeline items*, each having a unique identifier and a creation date as demonstrated in Figure 3.4. The last requirement for the remote model we demand is that at least one TimelineItem is the equivalent of our IssueComment, meaning that it contains a text body.

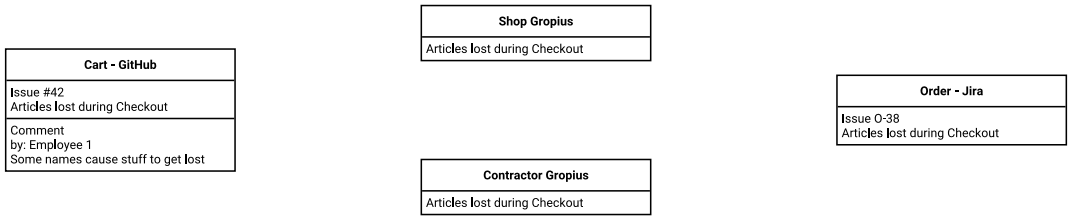
Conceptually the sync consists of two phases, first having an *Incoming* phase in which it acquires, processes and integrates the current state of the remote IMS, followed by an *Outgoing* phase in which it tries to modify the remote IMS to be as close as possible to the local changes.

3.3 Issue Replicators

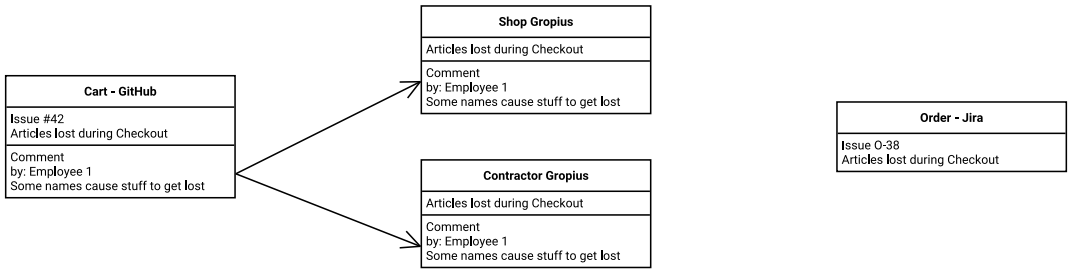
We usually lose all meta information about an issue once we sync it to an IMS. This can result in duplicates as different Gropius instances cannot communicate the chain of identical issues over multiple syncs.

For example, take the very common webshop example software with a “cart” service and an “order” service [BMCG20; HNSC21; WTEK20]. Now due to some bug in “cart” service, the “order” service does not get selected items. As this bug affects both the “order” service and the “cart” service, the developers put the issue in Gropius on both the “order” service and the “cart” service. As long as it is only one Gropius, here the Shop Gropius, the *Sync* can internally keep track of the IDs and thus does not duplicate any comments.

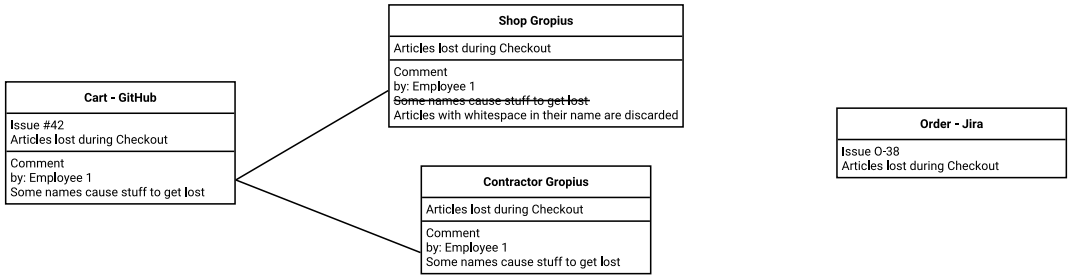
[1] https://ccims.github.io/gropius-backend-docs/login



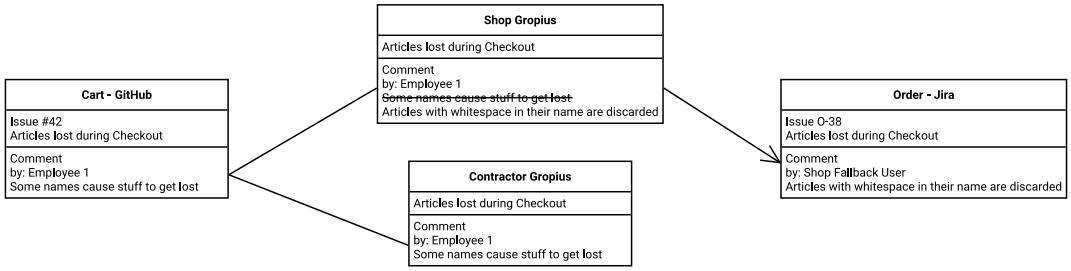
(a) Initial Situation.



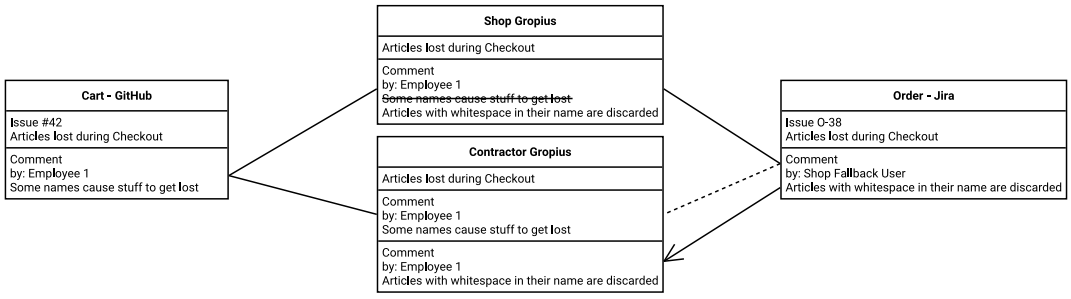
(b) First cycle of the *GitHub Sync*.



(c) Edit Comment in Shop Gropius.



(d) *Sync* of Shop Gropius.



(e) Incoming *Sync* Step of Contractor Gropius.

Figure 3.5: Start of an Issue Replicator.

After extending this example with a contractor who is supposed to write the “cart” service without us, we can construct a fully defective architecture. The contractor also pulls the issue onto both components in their Gropius instance. For example, assume someone creates a comment with mediocre helpful content on a cart issue Cart, which is maintained on Github as seen in Figure 3.5a. During the first *Sync* both Gropius instances simply sync the comment to gropius as expected, as seen in Figure 3.5b. While not strictly necessary for this example, assume that someone edits the comment in the Shop Gropius to a more sensible value, like in Figure 3.5c. Further slowing down time, assume that the Shop Gropius syncs first to *Jira*, sending the edited comment to *Jira* leaves us at the snapshot of Figure 3.5d. Now the fatal error occurs in Figure 3.5e: While trying to integrate the newly fetched information from *Jira*, the *Sync* now encounters a brand new comment. But from the perspective of the whole example we know that the comment is just an edited version of the one we got from *GitHub*, but the *Sync* has no way of knowing that.

We have worked out five solution approaches, described in detail later, but all of them share a few common properties: All of the following methods only work on the linking of issues, otherwise known as marking issues as originating from the same issue, not on the unlinking. Take for example two issues with accidentally identical titles and bodies: Once Gropius has synced these they are linked forever. If now one of the issues gets renamed and edited to hold the intended content, Gropius still considers it identical to the other one and forces these changes on the first issue. This results in the issues being synced until both hold identical content. For an example of the damage this creates: If two issues with identical titles and descriptions are merged together, but afterwards a maintainer marks one issue as a duplicate of the other results in Gropius marking the first issue also as duplicate, resulting in two issues which are closed as duplicates instead of one of them carrying the relevant content. In case the Gropius user has different permissions this can even be abused by intentionally creating duplicates: If a malicious attacker manages to get their manipulated issues marked as the same as a different one, they gain effectively all permissions the Gropius bot user has on the target issue.

In the following, we describe our five solution approaches in detail and discuss their limitations.

3.3.1 Invasive

The most secure method would be to tag every differing issue or comment with a unique identifier. Now if one comment should be integrated and another one with an identical tag already exists, it is guaranteed that both are intended to be the same comment. If both are tagged with different tags, Gropius has the guarantee that they are different comments. Only if one is tagged and the other one is not, any guessing is needed. This works identically with issues. This method requires that items be modified at least at some point during the *Sync*.

For marking issues we intend to use Universally Unique Identifiers (UUIDs), as they are per definition globally unique. We save this marking either in the title or description of an issue to identify issues and, at the end of the comment for comments. To avoid confusion with different content that matches the UUID pattern, we also enclose the raw UUID in square brackets.

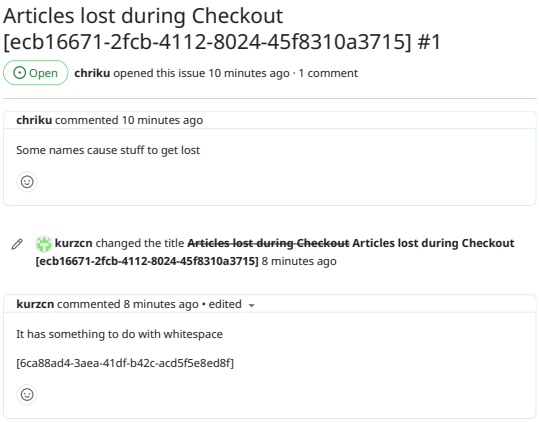
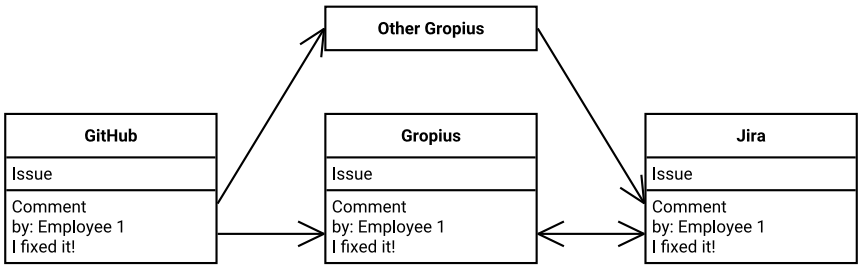


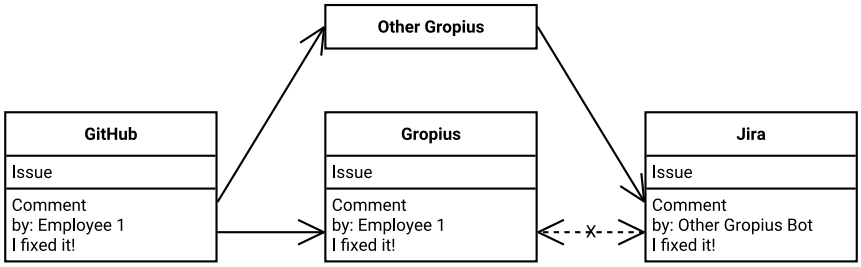
Figure 3.6: Example issue marked by invasive dereplicator.

The invasive dereplicator can either be used in an eager mode where it marks all issues as soon as possible as well as a lazy mode in which it marks issues only when syncing them as outgoing issues. The eager mode can lead to a more unified appearance in the issue tracker, which only marks issues when they are synced to somewhere else, which means that markings are mostly invisible for existing issues on the source IMS and only appear when Gropius creates issues.

3.3.2 Identical



(a) It works often.



(b) But fails even at small changes.

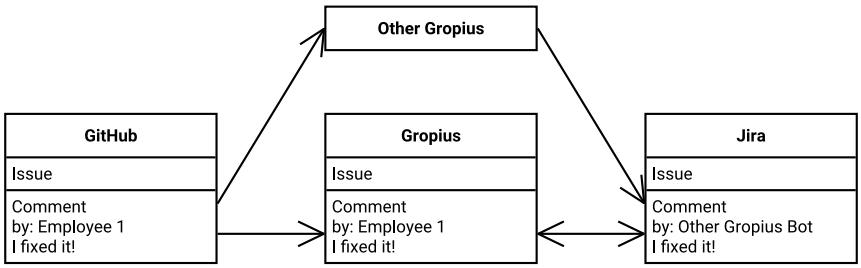
Figure 3.7: Identical Dereplicator.

Another simple, but noninvasive method would be to simply merge all identical items from different sources. While this would successfully prevent duplicate items from duplicating after the first time, edits as shown in Figure 3.5c can still introduce sporadical duplication. But as it only duplicates after user action, we still consider this a valid strategy to look at.

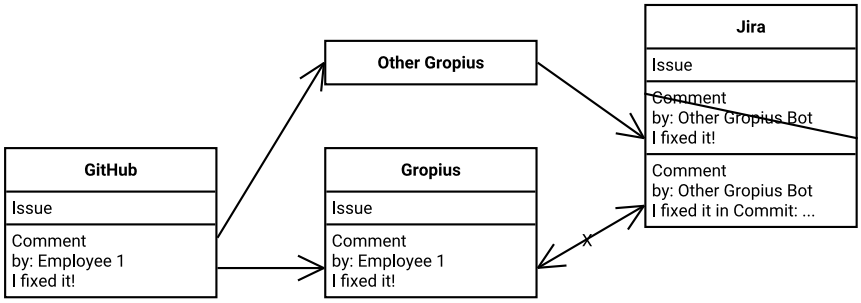
For issues, this means that during the incoming phase of the *Sync*, the *Sync* simply looks through all existing issues and looks for identical titles, authors and descriptions. If these match, the issues get merged. The process for comments is identical, just as there is no title, only content and author can be matched. As all dates are erased by the user-like API calls, considering them for matching is not relevant.

In the example of Figure 3.7a, we compare the author and the body of the comment, as these are identical to the comment we already know we mark it as connected and leave it. In the other example, namely Figure 3.7b, the user had to be replaced by the bot user and thus is different. Here we cannot notice it as identical and have to treat it as a new comment, this seems not noteworthy but as we have the issue now from both possible user accounts, this prevents us from duplicating it any further.

3.3.3 Heuristical



(a) Can work with small changes.



(b) But fails when all fields differ.

Figure 3.8: Heuristical Dereplicator.

Our heuristic dereplicator works similarly to *identical* but accepts minor changes. As the *Sync* often introduces small changes due to the limitations of API access, we set a threshold for which issues count as identical. For comments this would usually mean that the user is ignored and only the content is matched, as demonstrated in Figure 3.8a: While the bot user is a different user, as the body is identical we treat it as a duplicate and merge it together. For issues this can work

even better, as we have three fields guaranteed, that is title, author and description sometimes even additional fields can be matched to check for similarity. Of course, this does not work when all fields are different, as shown in Figure 3.8b. While the content seems similar, as this dereplicator only evaluates identical fields, no way of knowing that it has already a comment is possible. This results in a duplicate. It is important to calculate the threshold not evenly, different features have to be weighted fittingly: For example, a different author suggests a Gropius bot user, but different content with the same user is more likely to be a second, intended comment.

3.3.4 Natural language processing (NLP)

The heuristical dereplicator only considers identical fields, it is also possible to compare many fields a bit more precisely. For example, using NLP to check the similarity of bodies of issues allows changes that do not affect the meaning of the issue to contribute to the match. Text similarity may yield a score that describes the difference between two texts [II08]. This can be applied, for example, between the titles and descriptions of two issues to let the content weigh in with more precision than “equal” or “not equal”, similar to when used to detect duplicate issues [AHS13; HAS15; NNN+12].

However, within the scope of this thesis, we could not further develop this method due to time constraints.

3.3.5 Temporal

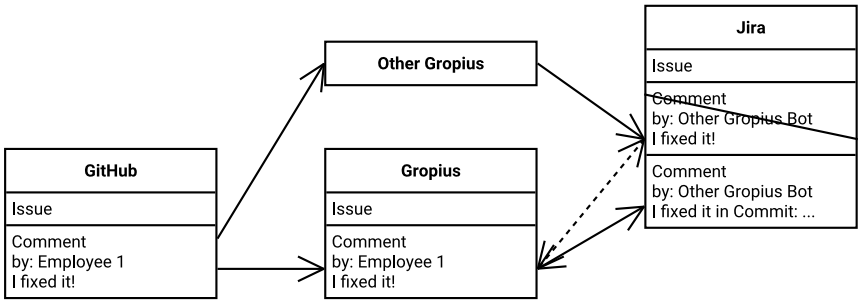


Figure 3.9: Temporal dereplicator can merge issues that have been the same.

Similar to heuristical, we compare the snapshots at different points in time with as much historical information as available to try and find matches where the issues may have been identical at some point in time. As most issue trackers and Gropius itself collect timelines for issues, this additional information should be used. Of course, this also does not account for lost information, like usage of the fallback user but should be able to work with higher thresholds due to not requiring both states to be in sync.

Another advantage of considering the timestamps is that a threshold can be implemented for issues to be similarly new. This can at least prevent some harmful issue takeovers, as crafting new issues to take over old ones has a limited time span where the attack is feasible.

feature	invasive	identical	heuristic	NLP	temporal
modifies issues	yes	no	no	no	no
merges identical issues with marker	yes	yes	yes	yes	yes
merges different issues with same marker	yes	no	no	no	no
merges identical issues without marker	no	yes	yes	yes	yes
merges similar issues with same spelling	no	no	yes	yes	yes
merges similar issues with different spelling	no	no	no	yes	yes
merges once similar issues	no	no	no	no	yes

Table 3.1: Overview of different dereplicators.

In the example of Figure 3.9 this means, that we see in the history of the issue tracker that there has been a version where the comments have been identical before being edited, so Gropius can mark the comments as same and transfer the change as a new change back into the timeline. As Gropius does not hold a comment history, this only works if the IMS-side comments are newer or for issues where both timelines are available.

3.3.6 Comparison

As seen in the table, the heuristical dereplicator is just a refined version of the identical dereplicator as the temporal dereplicator is the heuristic dereplicator with more comparisons. This results in them sharing the same flaws of being able to mistakenly identify duplicates and then merge them. The worst case can easily be reached with a misconfigured heuristical (or temporal) dereplicator: if the threshold is too low all issues are merged, resulting in Gropius trying to get all issues in this instance of the IMS to the most similar state possible. This would usually force a rollback (if possible) or manual labor to remove everything Gropius did as this would remove any sense of the issues.

A different danger coming from all three non-invasive variants is that they merge accidentally identical issues. This can be solved by the temporal and heuristic dereplicator by enforcing matching timeline items as additional markers. This can prevent issues with default or otherwise intended identical titles and/or descriptions from being merged. But this approach is vulnerable against repeating duplicates, for which an identical dereplicator modified to only stop after a certain number of identical issues are found can be used. This opens a whole new slew of combinations for these dereplicators to prevent more scenarios as they are thought up.

The invasive, marking dereplicator forces changes as soon as possible. While this means that it does not have the same issues as the non-invasive, changing all issue titles in an existing, public repo will usually have its own consequences. This makes this dereplicator useful for in-house projects but basically unusable for public projects with open development, as changing the title will probably not be allowed or seen as vandalism.

4 Architecture & Implementation

In this chapter, we lay out the architecture we want to build. First, we discuss the general architecture of the *Sync* in Section 4.1. Then, we dive deeper into the provided building blocks in Section 4.2. Finally, we round this chapter up with the two *Sync* phases, Incoming in Section 4.3 and Outgoing in Section 4.4. Our implemented version of this can be found at https://github.com/ccims/gropius-backend.

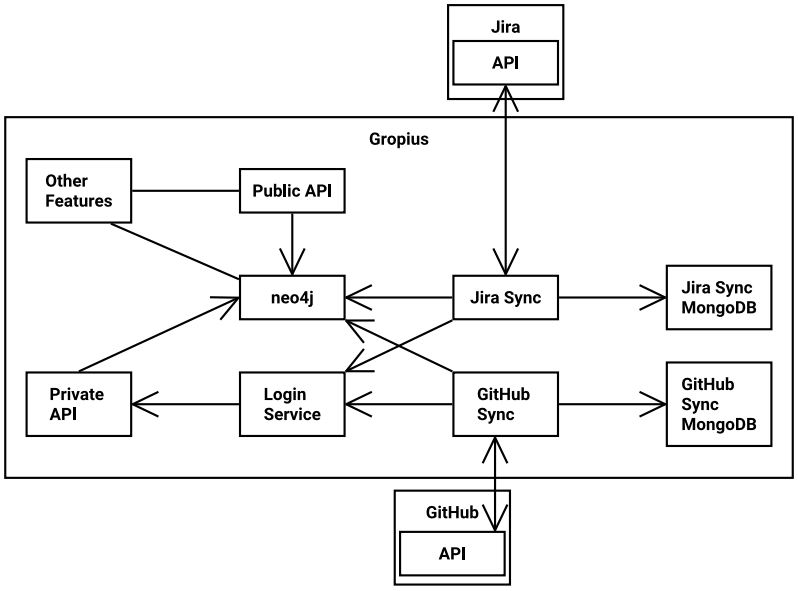


Figure 4.1: Architecture.

4.1 Sync Module

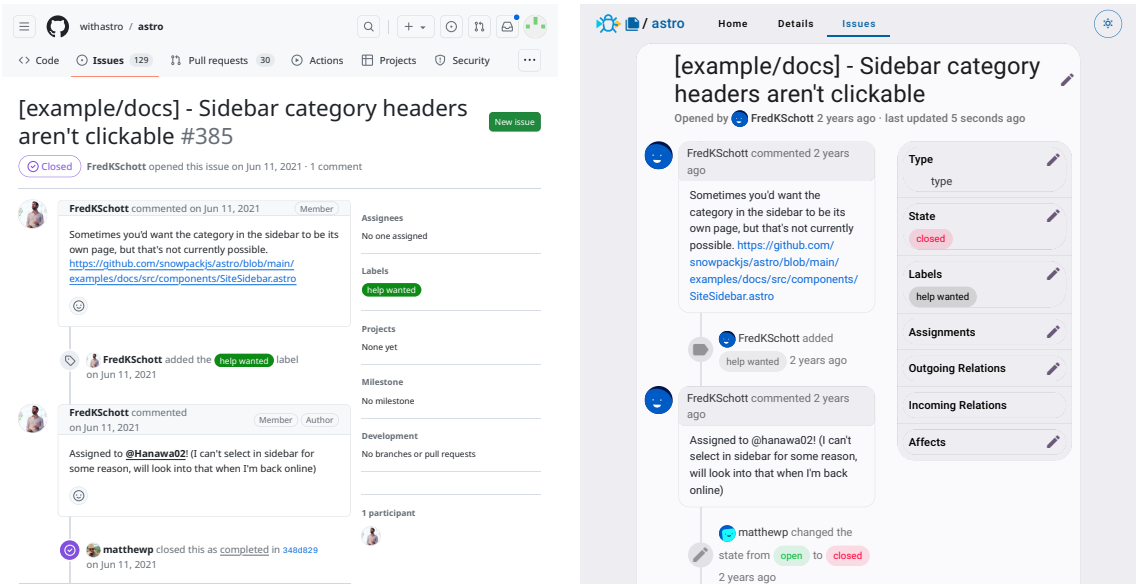
As syncing Gropius to different IMSs can be very different depending on the IMS, we have decided to follow a very modular approach. We make the *Sync* for each IMS a completely independent application and only premake building blocks to ease the creation of new sync modules [LSW87]. This allows full freedom in programming the main sync module while being sped up by allowing the usage of many premade building blocks that provide often-used functions. Thus each *Sync* can be built completely as needed, but in this chapter, we will address the intended way.

Each sync module has to interact with Gropius. For this two methods are theoretically possible: Either the use of one of the backends to provide a networked API or by directly accessing the database. As we already have structured access to the database using the classes that make up the

normal backend, we can simply import the Gropius data module and use it to comfortably access the database via GraphGlue. Furthermore, this allows us to access features that are usually blocked by the frontend, such as inserting old timestamps and avoiding restrictive permission checks.

Each sync module has to interface with the IMS in some way. We usually make use of the publicly provided APIs, for example, the *GitHub GraphQL* API and the *Jira* Representational state transfer (REST) API. This allows us to pretend to be normal users and access the data without having to request permission and avoid the resulting security dangers.

As a sync module needs to keep a complex internal state containing mappings between users, *timeline items* and sometimes even a whole mirror of the original repository, we recommend each module to use a separate database. For our building blocks, we use MongoDB which we have copied over from the previous *Sync* and recommend that the rest of the sync module should also use it as the database for *Sync* data. For the previous *Sync*, we decided on MongoDB, as it allows compact and comfortable storage from our codebase, while also providing a concise view of objects during debugging purposes. As these considerations have not changed with the new *Sync*, we see no reason to change it. Another consideration was to allow mixing the *Sync* data with the login service in one PostgreSQL database, but we specifically have decided against supporting that due to security concerns. While it would theoretically be possible to use the main Gropius database for our metadata, we have decided against that to avoid polluting the productive data with external data and ease the addition or removal of sync modules.



(a) Issue on *GitHub*.

(b) Issue in Gropius.

Figure 4.2: Example issue synced from *GitHub* to Gropius.

The *Sync* communicates with the login service for tokens required to access the API and to match the users found on an IMS to the users known in Gropius.

In summary, each sync module is a standalone application only interacting with its database, Gropius itself and the login service providing flexible development, maintenance and hosting capabilities.

Usually, the *Sync* is executed in cycles, which are started by running the applications. In the case of a cluster. this provides the ability to execute a sync module on a free node while also allowing the setup to be as simple as executing it in a timed loop.

The core framework tries to stay as close the the existing *Sync* implementation as possible. This means that conceptually most sync modules should consist of two halves: the incoming and the outgoing halve. The incoming part fetches the data and integrates it, while the outgoing part's job is to sync local data from Gropius to the IMS. This structure is recommended, as this is what the following building blocks were planned for.

4.2 General building blocks

All core components were written to support as many IMS as possible and are to ease the parts we think will be common during *Sync* implementation.

4.2.1 User Mapping

The User Mapping building block provides the mapping between a user-id, username or other identifier on the IMS side to one IMSUser in Gropius. This allows the behavior e.g. the combining and account searching for users with multiple IMS accounts for a given IMS to be centralized. This block will also contain the future integration with the login service, as soon as it is implemented. This mapping is stored locally for simplicity of use and is automatically kept in sync with Gropius. This module also provides a helper for the use of these users that for example keeps track of which user a comment has been created. In this example, if a comment has been created with the default user it usually can only be edited by the exact user and the *Sync* may not switch to the account on the IMS side, even if the user adds a token as the comment is owned by another user.

4.2.2 Labels Mapping

The Labels Mapping building block provides features for mapping remote labels to Gropius labels. It can be simply dropped in, filled with data and then mapped with the labels. It contains premade database access for the mapping and mostly just provides a safer and easier interface than using the database manually. This helps to normalize working with labels even if some info, for example, color, is currently unknown. While it does not do much work itself, it reduces the work needed for mapping labels to single function calls instead of having to work with at least one class for the database. For this, it provides a function that either takes an existing one or creates a new one given the IMS-side ID and all other available information. For reverse, it provides a function to look up the IMS-side ID or a given label or the advice to create a new one if no mapping is known. A function to register a newly created mapping is also provided.

4.2.3 Login Service Integration

The Login Service integration manages access to user tokens and other login service-related tasks. The main task is to get the closest token from the login service for a given user. While this is usually a simple task that only involves an IMSUser lookup, this can get often much more complex. As users can link multiple accounts together even at a later time, this also incorporates holding a mapping from previously used users for a given action to the current user. For example, if a comment has been synced to *GitHub* from the fallback user and someone later adds their *GitHub* account, the fallback user has to be used for edits to that comment as the true account has no permission to edit the comment. Another functionality this module implements is creating the IMSUsers if new users are encountered in the IMS.

As this block depends on interaction with the *login service*, which we were unable to implement due to time constraints, this block is not implemented in a usable fashion.

4.2.4 Notification Handling

In case of an error, an involved user has to be notified that the issue has errors during sync. The potential errors can range from a simple authentication error if a user has connected an invalid token to disk full for the *Sync* database to errors in the *Sync* itself. As a notification system for Gropius is currently only in the earliest stages of planning, this results in saving the message in specific templated files for a frontend to retrieve. As soon as frontend implements displaying this information, this should result in visible notifications on either the issue or the project depending on the specified target.

4.2.5 Template Matcher

The *Sync* needs an entry where to search for IMSProjects that could be synced. For this, the *Sync* searches first all IMSs that can represent it. To do this there is the Template Matcher component ready-made to search through all templates. It takes a set of template specifications (one each of IMSTemplate, IMSProjectTemplate and IMSUserTemplate) and then searches for all connected IMSTemplate, IMSProjectTemplate and IMSUserTemplate which can be interpreted as the specified template. The implementation checks for identical titles to ensure that only templates for the right *Sync* can be matched. Currently, the “specified” means only identical templates, but as soon as Gropius has the logic for editing the type of templated fields, the function should support edited templates that describe more specific fields than the current templates. If no templates are found, it automatically creates a fresh set of templates using the original specifications to allow users the easy and comfortable creation of new IMS.

4.3 Incoming

During the incoming phase, two main steps are usually performed: Fetching new data and integrating the new data into Gropius.

4.3.1 Fetching

The *Sync* has to request all new data since the last sync cycle. For this, we wrote a few helper classes to help with page-based APIs, but otherwise, the fetching step is too dependent on the peer IMS to generally help with.

Paginator

As most of these APIs are made for human consumption, data is usually provided as pages. Pages mean that a specific number of items of the result are grouped, usually with the intention that the user only gets displayed this subset and can switch to different pages. As we usually need all items of a query, we provide a Helper to automatically iterate through all pages using the Load Balancer. We do this by implementing it for a *cursor*-based API, meaning an API that returns the *cursor* which can be used to retrieve the next page. Usually page-based APIs can be handled like *cursor*-based APIs with the cursor for the next page being the current page added to the size of the current page, which is usually either one or the number of items on the page.

Load Balancer

As we talk as API consumer to the IMS, we usually have to observe rate limits. The Load Balancer building block works on queries, which have an identifier that should stay the same even over multiple sync cycles, an executor, a base priority and an estimated cost. It allows simply specifying a list of queries and does the scheduling depending on rate limits and estimated query costs. Currently, we implement a simple priority-based scheduler, which increases priority for not executed queries for the next sync cycle by saving it in the database, but intend to upgrade it to the practical specifications derived from future work. We currently do not support adding new items to the queue after starting, resulting in e.g. new issues needing a second sync cycle for the content to appear.

4.3.2 Integrating

During the integration step, all of the fetched data has to be integrated into Gropius. As Gropius is flexible enough, this results in simply adding the new *timeline items* to the timeline and then recalculating the new properties. As described in Section 2.1.4, we do this by applying all *timeline items* on all modified issues in order of creation.

Before inserting a *timeline item* into the respective issue and before creating an issue, we attach the main entry hook of the dereplicators here: Before saving the issue to the database, we select the corresponding dereplicator as specified on the IMSProject and give it the issue. Invasive dereplicators can now insert *timeline items* that insert the required tags or other needed data. Afterwards, we ask the dereplicator if we should insert *timeline items* or create the issue. In case the dereplicator disallows adding, because it has found a previous match the new issue is a duplicate of, we add the remote issue as synced and link it to the original issue. Identically for *timeline items*, we simply add another remote to our mapping causing it to be synced to this one if new information is available in the future.

4.4 Outgoing

The outgoing phase is usually less clearly separated into steps. The usual steps for syncing outgoing data are collecting items, either *timeline items* or *issues*, that need to be synced, sending these items and then marking the successfully synced items as synced. These steps may sometimes be easier if applied to each item before starting the next.

4.4.1 Collecting

The Collecting phase searches for items that need to be synced. This is as simple as finding all issues without remote ID and of the existing issues all *timeline items* without known IMS counterpart of conflicting TimelineItem. Conflicting TimelineItems happen, for example, if someone adds a label and then immediately removes it without a sync cycle in between. In this case, we judged it unnecessary to add a label on the remote just to remove it, as that would clutter their timeline unnecessarily and not provide any practical information. This may also catch accidental label additions, as long as they are noticed before the next sync cycle. This system does not influence anything that stayed a full successful sync cycle, as it only checks the most recent, unsynced items.

4.4.2 Pushing

The second extremely *Sync*-specific section, the pushing phase just transfers the changes to the remote IMS and is expected to consist mostly of IMS-specific network calls.

This usually requires interaction with the load balancer from the Incoming Step, as described in Section 4.3, as the quote is expected to be shared between these steps, thus any rate limit notifications need to be forwarded and combined to ensure being able to push changes to the IMS. If this interaction does not happen, the *Sync* will continue using up the API budget during the Incoming phase and starve the Outgoing phase.

4.4.3 Marking

After an item has been pushed, marking is usually as simple as setting the remote ID of the given database entry which usually will be queried during the write. This step contains very important error handling, as errors here can involve complex interactions with different systems. For example, a “permission denied” error when syncing an item may need to notify the Login Service that a given token is invalid. Contrary an error for reaching a rate limiter quota may need to inform the incoming load balancer to reduce the amount queried, to push out the changed data. Differently from the incoming error handling, mistakes here will lead to unfixable remnants on the timeline of IMSs, which should be avoided whenever possible as this could confuse unaware users and may even lead to being restricted due to claims of vandalism.

4.5 Sync Adapter

We implemented both sync adapters as separate projects.

4.5.1 GitHub

GitHub's model was a main inspiration for Gropius and is thus very similar. While issues consist of *timeline items* and current values, reading only *timeline items* and a base state can successfully be used to reconstruct an issue, as described for Gropius before. This model does offer very limited flexibility but makes it up by the number of *timeline items* which cover most situations.

We were able to transfer much code from the existing *Sync* implementation. This resulted in us being able to take the fetch step, break it into pieces, and add it into the load balancer. For this, we balance a query for the list of issues as well as a timeline and a comment query for each issue that has not been marked as finished since the last change. The comment query as well as the cautious syncing is necessary as *GitHub* can be asked to give *timeline items* after a given point in time, but this does not include comments changed after that point in time. Thus the *Sync* has to query each issue with a new date for each comment to check if it may have been changed. While this results in a huge overhead until the issues have been marked as done, it is necessary to not miss updates in comments.

4.5.2 Jira

Jira instead has basically no fixed content in an issue and instead makes most things part of fields. This means that most actions on an issue just change fields of an issue, which are then saved in the *changelog*. This also allows the reconstruction of issues from a list of comments and field changes but keeps most data generalized as fields.

The *Jira Sync* was also relatively easy, as the changelog items have unique IDs and can thus be simply mapped by the field they change. The main issue for our *Sync* for *Jira* came from the fact that it does not return the changelog ID after doing a change, this means that during the next *Incoming* phase, the *Sync* does not know which *changelog* entries are caused by a given action and which were done separately. This results in the *Sync* being unable to mark a given *timeline item* to be connected to the corresponding changelog item, which currently results in the *Sync* duplicating every comment synced to *Jira*. While this may sometimes be handled by dereplicators, the fact that the fallback to bot user may change the author of the comment or *changelog* item, does also not have a perfect solution.

4.6 Dereplicator

We hooked the dereplicators into the *Incoming*, more precisely the *Integrating* part of the issues. If it finds the original issue, we can then merge it together otherwise leave the ID null to let GraphGlue create a new node for us. We then implemented both base dereplicators as base classes of the

interface, with the invasive variant just inserting a new *timeline item* that changes the title. The heuristical dereplicator takes the thresholds as arguments, which allowed us to implement the identical dereplicator as an inheriting class that sets the parameters to 1.

5 Evaluation

In this chapter we examine our four research questions: “RQ 1: Can the *Sync* keep a stable copy of a project?” in Section 5.1, “RQ 2: Can issues be synchronized between multiple IMS with different models?” in Section 5.2, “RQ 3: Which issue dereplicator can work on an external open source project without false positives?” in Section 5.3 and “RQ 4: Can issues be identified as originating from the same issue after reducing them to their IMS representation?” in Section 5.4. We then examine the results of all these combined in Section 5.5 before examining the threats to the validity of our answers in Section 5.6.

5.1 RQ 1: Can the *Sync* keep a stable copy of a project?

The main function of the *Sync* is to keep the issues of an IMS synchronized with the issues on Gropius. We define a Gropius issue as similar to an issue on an IMS if a stakeholder could work with both versions without getting interference from our *Sync*. To work with an issue effectively, no information may be lost or false. We compare all supported parts of the issues, which are titles, comments, labels, and, the open/close state. As our *Sync* is only eventually consistent, due to API limits and the possibility of editing the issues while the *Sync* is running, we have to introduce a time component after which this may be true, which due to the current implementation is done if the *Sync* has the resources available and can run without other obstacles for two full successful invocations for a given issue. Also, we have to go around the rate limits themselves as we cannot work with the IMS faster, resulting in us having to use “full successful sync cycles” which we define as full successful invocations without error or running into rate limits. We have to give ourselves here a larger margin for error by increasing the number of sync cycles to three, to allow the *Sync* to handle the setbacks given by running into rate limits and other obstacles. We also enforce the result to be stable, meaning that the *Sync* neither needs to do nor does any changes to the issue once all relevant information has been copied over until new changes are introduced to the issue. While we currently do not utilize it, we consider changes on the issue level, meaning that a single changed timeline item may carry additional changes to other parts of the issue. This results in our hypothesis being that *H1*: “A synced project contains equivalent issues/timeline items as the original after three full successful sync cycles”.

To test this we choose a few suitable repositories on both *GitHub* and *Jira* and sync each project to a local Gropius instance into different, independent projects. For each project, we do this three times, once for each implemented dereplicator configuration. We currently have to limit ourselves to smaller projects as Gropius itself does currently expose multiple bugs when going above an unknown and content-dependent number of issues, which we estimate is somewhere around 4000 issues in our cases, after which the backend did not allow us to view the issue list, which we have been assured will be fixed soon, and the *Sync* itself was outside of tolerable initialization times, which we set to 6 hours for this project. As our first *GitHub* project, we took a relatively new web

framework called *astro*[1], as they have a sizeable number of issues, at the time of writing more than 3500, and had on average 4.6 issues per day during the last month. As the second project, we chose *Mojang Web*[2] due to being relatively sizeable, hosted on *Jira* and containing many very similar issues to due management policies. For this project, we had to limit our *Sync* to the first three pages of issues to comply with rate limits and limit the number of issues to something reasonable. Finally, we chose two other projects with high activity, named *mojo*[3] and *bun*[4] due to the high rate of new issues while still being in our range of working issue counts. We then instructed the *Sync* to sync each project to our Gropius instance for 24 hours. As the *Sync* runs in cycles, we first run the *Sync* in intervals forced upon us by the rate limit until we have imported the existing data and afterwards switch over to running it periodically with 10 min pause, which we think is a reasonable delay until a stakeholder can be expected to know about new events on the IMS, between runs. We deem that a reasonable compromise between faster updates and more API usage. Afterwards, we compare 20 randomly selected issues as well as all new and edited ones since starting the experiment between the original on the IMS and the synced ones in Gropius according to the previous criteria. Incompletely synced issues, meaning issues that have been edited after the third to last sync cycle are ignored for this experiment.

5.1.1 Results

After running the *Sync* through the initial passes, many issues were visible in the project overview. The number of issues, when synced with the identical dereplicator also matches the number displayed on the source IMS for each project, excepting the lowered number for the limited project. With the heuristic dereplicator, we had fewer issues than expected, this being 50 of 9121 missing for two identical features and 7163 missing with one identical feature which we further researched in Section 5.3. These issues were missing since the first full successful sync cycle of the experiment and were not synced, no issues vanished once they had been on Gropius. Each compared issue matched the original version, except for knowingly unimplemented features. None of the falsely dereplicated issues was under our compared issues, but we examine one of the broken issues in Section 5.3. Watching for further changes, no unexpected changes were observed, the only changes happening are content changes and new, but empty issues after the next sync cycle with the content for these empty issues appearing one sync cycle later.

5.1.2 Discussion

All in all, we think the *Sync* can function for this task with the identical dereplicator. Of course, this does not absolve it from the yet unimplemented features, as working with the bare minimum is not ideal. But the implemented features work as intended, we hold a stable copy of the source. With this, we accept the hypothesis, that our project contains equivalent issues/timeline items as the original after three full successful sync cycles. Thus we answer **RQ 1**: “Can the *Sync* keep a stable copy of a large project?” with a clear “yes”.

[1] https://github.com/withastro/astro
[2] https://bugs.mojang.com/projects/WEB
[3] https://github.com/modularml/mojo
[4] https://github.com/oven-sh/bun

5.2 RQ 2: Can issues be synchronized between multiple IMS with different models?

A second integral function of the *Sync* is to map between different IMSs. As these IMSs have different models two issues cannot be easily compared, thus evaluating the similarity is not trivial. Thus we evaluate it based on the perspective of a user: We try to find out if all information that has been put into the original issue is also represented in the synced issue. For this, we consider only user-modifiable information, excluding for example dates and users to evaluate as similar as possible to if a user had tried to recreate the issue manually. Also irrelevant for our use case is internal management information, thus we evaluate purely on what is presented to the user using the normal User interface (UI). Combining that gives the hypothesis *H2*: "Every user-facing, user-modifiable information written in the original issue has after syncing an equivalent information in the destination issue"

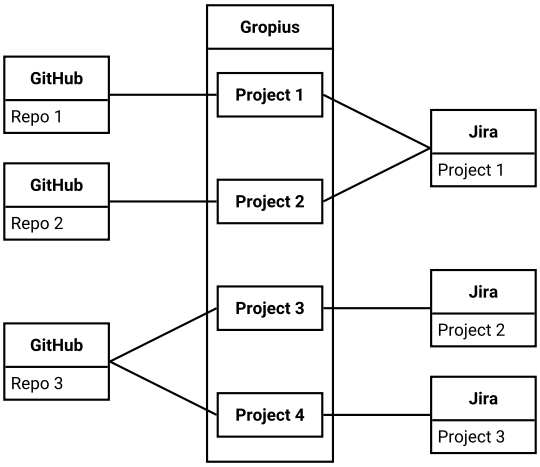


Figure 5.1: *Sync* Setup.

To evaluate this, we set up two independent projects: The first one syncs one project from *GitHub* over Gropius to *Jira* and with a different project in Gropius to a different project on *GitHub*. The second one does the same but with *GitHub* and *Jira* switched. We then create issues with differentiable inputs for all user fields allowing user input and comparing the original issue, after the *Sync* has stabilized, with the synced one. We disable all dereplication features, as this experiment is only intended to test the model conversion.

5.2.1 Results

Currently, most operations the *Sync* supports, this being titles, descriptions, and comments use a similar enough model in Gropius, *GitHub* and *Jira*, which allows the *Sync* to convert them losslessly. As assignments are not fully implemented in the *Sync*, they were not synced at all. Labels on *GitHub* are nodes with specific IDs, while on *Jira* they are just texts. If multiple, identically named labels were available on *GitHub* the *Sync* did not necessarily add the same label as the original but used any similarly named label on *GitHub*. Similar to that, duplicate identical labels were lost as well. On the other direction, as all *GitHub* labels contain a name that contains everything that

makes a *Jira* label, syncing labels from *Jira* over *GitHub* to *Jira* did not lose any information. The open/close state of the issues was not synced to *Jira* and stripped down to only the bare open/close value when going from *Jira* to *GitHub*. As is a known bug due to the *Jira* API, many timeline items synced to *Jira* were duplicated, but have been deleted for this experiment, as this issue should be fixed in the near future. Other features on *GitHub* and *Jira* are currently not implemented and thus are not synced.

5.2.2 Discussion

While we deem the current functionality good enough for basic usage, we currently have a great amount of loss when using advanced features. Aside from the assignments, which we had to cut due to time constraints, our main worry is the open/close state. While *Jira* enforces a strict workflow an issue has to go through, *GitHub* does not. Even assuming the mapping to be done, on *GitHub* simply closing an issue is always a possibility, which may not be in line with the *Jira* issue workflow. The *Sync* would have to transition the issue through possibly multiple states to reach a state that matches the state on *GitHub*. Otherwise, most features are mostly for convenience and we can imagine working with issues without them, so while they would be nice to have do not block usage of the *Sync*.

As this only mentions implemented features, we are far from being able to sync every user-facing or user-modifiable information in an issue, forcing us to reject the hypothesis. We have ideas to use comments to document timeline items unsupported by the target IMS, but these are future work. The open/close state may be approximated, by knowing and following the workflow graph or can be mitigated manually by changing the workflow to allow state changes, which may disrupt the normal work. In conclusion, we do not see it as possible to fully sync everything perfectly between the models as imagined in RQ 2, but it should be possible to come close enough for practical usage with much more development work.

5.3 RQ 3: Which issue dereplicator can work on an external open source project without false positives?

As previously established in Section 3.3, false positives in the dereplicators can be harmful, knowing the expected rate is necessary to trust the *Sync* to not damage repositories. A false positive in this case means that a dereplicator merges two issues that were not intended to be the same issue. To estimate practical conditions, we consider projects that do not have any knowledge of Gropius and practically work with the issues they create. This allows us to estimate the failure rates when used for practical results, instead of laboratorially crafted issues that specifically target the known weaknesses of our concepts. Resulting in the hypothesis *H3*: “It is possible to design an issue dereplicator that does not erroneously merge different issues”.

For this, we modify the *Sync* to log errors instead of merging issues when a dereplicator finds a match. As all our known replicators need multiple connections or duplicate issues, we assume that normal activity on these issue trackers does not contain any automatically duplicated issues. If any dereplicators find matches anyway, we categorize it as a falsely dereplicated issue. We are not willing to break existing projects, this test can only be used with non-invasive dereplicators,

as we neither have permission to nor would be willing to vandalize all issues by unnecessarily editing every issue. We have decided that missing out on the invasive dereplicator does not change anything, as it would qualify all newly encountered, markerless issues as different anyway. As these additions only write the attempted actions to a log and do not change any behavior of the *Sync*, we introduce these small additions into the experiment from Section 5.1. In the log, we have the identical dereplicator and the heuristic with thresholds of one and two. The thresholds are the number of features compared, meaning that on two more than two items out of the title, the author and summary have to be identical for an issue to be merged.

5.3.1 Results

The identical dereplicator did not match any issues. Further investigation is that in the *GitHub* projects most issues had a different title. Contrary, the *Mojang Web* had many issues sharing the title "Delete my account", but always had different authors. Combined all repositories, only 0.5% of all issues shared two features with a previous issue, while 78% shared at least one feature with a previously synced issue.

5.3.2 Discussion

As seen from LLVM, with a moderated project and high-quality issues, comparing the title and author is enough to avoid any false positives. On the other hand, having a policy like *Mojang* that creates many similar issues leads to us confusing them together with a threshold lower than three, which would include any issues created in the future and thus merge them into one gargantuan issue. This would not only allow a malicious attacker to get the *Sync* to create an enormous amount but also lead to multiple copies of the one merged issue in accidents. Of course, the invasive dereplicator is as vulnerable, as using a marker that has already been used can have the same effect. But restricting it to identical issues does not lead to false positives while still restricting the *Sync* from syncing already duplicated issues, as we have seen in Section 5.2 that the *Sync* creates as identical as possible issues and all unrepeatable differences have been already removed during the first pass. As we have found a, albeit limited but safe method, we can accept *H3*: "It is possible to design an issue dereplicator that does not erroneously merge different issues". To conclude this, we can answer **RQ 3**: "Which issue dereplicator can work on an external open source project without false-positives?" with the identical and the invasive dereplicator.

5.4 RQ 4: Can issues be identified as originating from the same issue after reducing them to their IMS representation?

This question assumes the following setup of Gropius instances and projects on IMSs: Assume at least one Gropius instance has already synced to multiple IMSs. Each issue on the IMSs may have an issue on each Gropius instance, but each Gropius issue can sync to multiple issues on the IMSs. Now the question is if a different, newly setup Gropius instance with no knowledge about the other Gropius instances can piece together which issues on the different IMSs are represented in the original Gropius instances by the same issues and which are different issues. With this, we

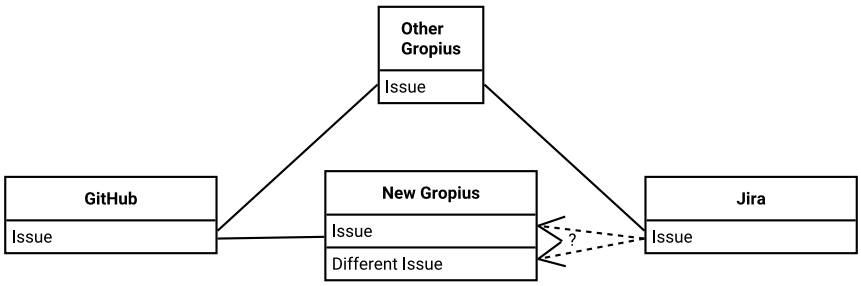


Figure 5.2: Example questionable issue flow.

arrive at the hypothesis *H4*: "A new Gropius instance syncing from an existing set of IMSs with projects synced to by different Gropius instances can identify which issues are intended to be the same issue."

We test this by setting up two Gropius projects and one project on each of the two supported IMSs. We then create issues matching the implemented timeline items and the settings of the dereplicators in the first Gropius project. As for the issues created, we include two with the same marker in their title. The first is for the invasive dereplicator to identify them as identical, and the second is to examine the damage caused by erroneous marking. A third issue with an identical title, but a modified description is used to attempt to trick the heuristic dereplicator into an erroneous merge. After the creations, we let the *Sync* run for this project until it has stabilized to sync the issues onto the IMSs. Once the issues are confirmed present we run the *Sync* on the second Gropius instance with each dereplicator configuration we select and then check if it has been merged or has been synced as two issues.

5.4.1 Results

As expected, the invasive dereplicator merges both issues with the same marker and combines them, even though they were different issues on the original. The heuristic dereplicator and the identical dereplicator merge the issue with identical titles, descriptions and authors. Also, the heuristic dereplicator on settings of two out of three and below also merges the one with the changed description into the mixed issue.

5.4.2 Discussion

As seen in the experiment, the dereplicators can be easily tricked into merging issues together. We have not found any kind of implicit marker to identify issues safely as same or different, only if we explicitly add one ourselves into visible fields we can uniquely mark issues. Thus we have to reject *H4*: "A new Gropius instance syncing from an existing set of IMSs with projects synced to by different Gropius instances can identify which issues are intended to be the same issue." and try to approach the solution using statistical and unreliable methods. We lose too much information about the issues, so **RQ 4**: "Can issues be identified as originating from the same issue after reducing them to their IMS representation?" currently does not seem possible.

5.5 Discussion

In conclusion, we think we have made a workable base for future *Sync* versions. We can sync most basic features which we think allows working with a project through the *Sync*. Of our dereplicators, only the identical dereplicator did not falsely merge issues together. The heuristical dereplicator, with both thresholds, combined issues erroneously, which would have damaged the issues in question on the IMS if outgoing had been enabled. All dereplicators stop issues from replicating infinitely, so we judge it reasonable to run on servers without continuous supervision. Overall, until something new has been found, we recommend using only the identical dereplicator and manually dealing with duplicates that were not caught by the dereplicator. On the other hand, adding the missing features, not just including the missing timeline items but also the support for mapping unsupported timeline items to comments for IMS that lack native support was outside the scope of this thesis. While we were able to lay much-needed groundwork for future versions of the *Sync* and provide a codebase to develop from, we are not able to provide a *Sync* that can be used without limiting the users. We hope that future work can fix all of these issues and maybe even find a solution for issue replication.

5.6 Threats to Validity

There are four aspects of validity according to Runeson and Höst [RH09], these being construct, internal, and external validity, and reliability. For internal threats, we only tested both parts, the *Sync* and the dereplicators, together and thus does not know if the dereplicators work differently with another implementation of the *Sync* or if the dereplicators affected any result of the *Sync* itself. But as we are not aware of any other Gropius *Sync* and tried to generalize the results over the dereplicators, we do not think it has affected any results, except the ones that are describing our implementation anyway. We had to limit our choice of test repos to match the current abilities of the involved systems, which due to the current performance of the *Sync* and per-query memory limitations of the backend meant that we had to specifically search for high-activity but low issue count projects, which lead us to mostly test on the issues of *JavaScript* frameworks. While this may present an external threat to validity, we still think these projects are managed differently enough to generalize our results. For the other experiments, we only generated the test issues ourselves, but as the only requirement that affected the results experiments was to touch all fields, we do not think that there is any chance different issues change the results. As for construct threats, we did not ensure that all dereplicators always got exactly the same data, as issues may have been edited during different parts of the runs, leading to minor discrepancies. However, even though we tested with a high number of issues overall, the results included a very limited number of edits. Therefore, we think the influence of these edits may have had on the resulting numbers is negligible. We only compared issues for identicality ourselves, which may be a threat to the reliability. As we have declared the relevant parts identically, we do not assume that any understanding different enough to notably change the results is possible.

6 Conclusion

6.1 Summary

We were able to build a set of base modules that can be used to build new sync adapters with much more ease than starting at nothing. We used these modules to provide two sync adapters, one for *Jira* and one for *GitHub*, both proving the concept as well as uncovering challenges that have to be solved in the future and can be planned within future modules. The dereplicator problem, while not solved fully, has been reduced to a state where creating a *Sync* topology can be built that does not replicate issues infinitely, thus allowing the *Sync* to be run with less supervision. We evaluated multiple approaches to dereplicate issues, giving an initial overview of what we can expect from dereplicators and where the limits should be expected.

6.2 Benefits

We were able to provide Gropius with a functioning *Sync*, which should allow further evaluation of all of Gropius as a more complete project. Instead of relying solely on manually created test data, now it should be possible to just sync in external projects, allowing software architects to relate issues and software developers to link duplicates together.

6.3 Limitations

Our *Sync* currently only supports basic features and is thus unable to handle more complex features, like assignments, that are expected of it. This limits the *Sync*'s usage in practical applications, as not even seeing missing items may lead to wrong conclusions when looking at issues. Furthermore, *Jira* has limited functionality due to time constraints, making issues synced to *Jira* stuck in the open state with the current *Jira Sync* being unable to walk the state graph. This greatly limits even basic usage of the *Sync* for practical projects, as marking issues as closed is necessary for keeping an overview of all issues.

For dereplicators, we were unable to implement and thus evaluate the more complex NLP and temporal dereplicators and thus cannot give any estimation about their practicality. Also as Gropius has currently unavoidable limits to the issue count we were only able to evaluate on a comparatively small dataset.

6.4 Lessons Learned

We greatly underestimated the time it would take to integrate a completely different IMS, in this case, *Jira*. We expected a much easier task, as with *GitHub* and did not note how much Gropius was designed to be similar to *GitHub* and the amount of ease this introduced for syncing to *GitHub*. For example, we copied *GitHub*'s lax issue-state model, of which each state can transition to any other state with a single action, while *Jira* has a workflow, that specifies the possible transitions, which we have to follow if we want to transition an issue to another state using the *Sync*.

We were unaware of how niche the use case of our more complex dereplicators is, as they do more harm by merging different issues instead of finding duplicates. We expected real-world issues to be far more varied and thus created elaborate, but irrelevant use cases during planning which the dereplicators may solve but do not happen outside of laboratory conditions.

6.5 Future Work

To properly use the *Sync*, it is necessary to add many of the missing *timeline items*. Furthermore, due to the state of the Gropius ecosystem, we were unable to implement the connection to the login service, which needs to be added to allow the *Sync* to simulate being different users to the IMS. Our implementation was heavily influenced by our management of the limited time, resulting in the codebase needing more work and testing before being used for more complicated tasks.

For the issue dereplicator problem, we were able to find methods to contain uncontrolled, infinite duplication but struggle with the limited duplication of issues the first time, until the *Sync* made an identical copy of the issue. We hope that future work on the topic of dereplicators is able to somehow find a way to reliably merge issues that have originated from the same issue without causing as much potential damage. We see our work as a foundation and hope that future work may be able to build something great on that.

Bibliography

[AHS13] A. Alipour, A. Hindle, E. Stroulia. “A contextual approach towards more accurate duplicate bug report detection”. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, May 2013. DOI: 10.1109/msr.2013.6624026 (cit. on p. 18).

[AMR20] J. A. Añel, D. P. Montes, J. Rodeiro Iglesias. “Cloud and Serverless Computing for Scientists: A Primer”. In: (2020). DOI: 10.1007/978-3-030-41784-0 (cit. on p. 4).

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. “Portable Cloud Services Using TOSCA”. In: *IEEE Internet Computing* 16.03 (May 2012), pp. 80–85. DOI: http://doi.ieeecomputersociety.org/10.1109/MIC.2012.43 (cit. on p. 6).

[BJS+08] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, T. Zimmermann. “What makes a good bug report?” In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. SIGSOFT ’08/FSE-16. ACM, Nov. 2008. DOI: 10.1145/1453101.1453146 (cit. on p. 3).

[BMCG20] B. Benni, S. Mosser, J.-P. Caissy, Y.-G. Guéhéneuc. “Can microservice-based online-retailers be used as an SPL?: a study of six reference architectures”. In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. SPLC ’20. ACM, Oct. 2020. DOI: 10.1145/3382025.3414979 (cit. on p. 13).

[BVGW10] D. Bertram, A. Voida, S. Greenberg, R. Walker. “Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams”. In: *Proceedings of the 2010 ACM conference on Computer supported cooperative work*. CSCW ’10. ACM, Feb. 2010. DOI: 10.1145/1718918.1718972 (cit. on p. 5).

[Crn03] I. Crnkovic. “Component-based software engineering - new challenges in software development”. In: ITI-03 (2003), pp. 9–18. DOI: 10.1109/iti.2003.1225314 (cit. on p. 4).

[CV85] R. E. Crable, S. L. Vibbert. “Managing issues and influencing public policy”. In: *Public Relations Review* 11.2 (June 1985), pp. 3–16. DOI: 10.1016/s0363-8111(82)80114-8 (cit. on p. 3).

[DW07] A. Dubey, D. Wagle. “Delivering software as a service”. In: *The McKinsey Quarterly* 6.2007 (2007), p. 2007 (cit. on p. 5).

[HAS15] A. Hindle, A. Alipour, E. Stroulia. “A contextual approach towards more accurate duplicate bug report detection and ranking”. In: *Empirical Software Engineering* 21.2 (June 2015), pp. 368–410. DOI: 10.1007/s10664-015-9387-3 (cit. on p. 18).

[HNSC21] S. G. Haugeland, P. H. Nguyen, H. Song, F. Chauvel. “Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps”. In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2021, pp. 170–177. DOI: 10.1109/seaa53835.2021.00030 (cit. on p. 13).

[II08] A. Islam, D. Inkpen. “Semantic text similarity using corpus-based word similarity and string similarity”. In: *ACM Transactions on Knowledge Discovery from Data* 2.2 (July 2008), pp. 1–25. DOI: 10.1145/1376815.1376819 (cit. on p. 18).

[JFC+09] M. R. Jakobsen, R. Fernandez, M. Czerwinski, K. Inkpen, O. Kulyk, G. G. Robertson. “WIPDash: Work Item and People Dashboard for Software Development Teams”. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 791–804. DOI: 10.1007/978-3-642-03658-3_83 (cit. on p. 8).

[La17] S. La. “Defects management in Embedded Systems”. In: (2017) (cit. on p. 8).

[LSW87] M. Lenz, H. Schmid, P. Wolf. “Software Reuse through Building Blocks”. In: *IEEE Software* 4.4 (July 1987), pp. 34–42. DOI: 10.1109/ms.1987.231062 (cit. on p. 21).

[MK11] S. Mahmood, A. Khan. “An industrial study on the importance of software component documentation: A system integrator’s perspective”. In: *Information Processing Letters* 111.12 (June 2011), pp. 583–590. DOI: 10.1016/j.ipl.2011.03.012 (cit. on p. 4).

[MNH15] S. Mahmood, M. Niazi, A. Hussain. “Identifying the challenges for managing component-based development in global software development: Preliminary results”. In: *2015 Science and Information Conference (SAI)*. IEEE, July 2015. DOI: 10.1109/sai.2015.7237254 (cit. on p. 5).

[Neu20] T. Neumann. “IDE Support of Issue Management for Component-based Architectures”. Bachelor’s Thesis. University of Stuttgart, 2020. DOI: 10.18419/opus-11608 (cit. on p. 6).

[NNN+12] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, C. Sun. “Duplicate bug report detection with a combination of information retrieval and topic modeling”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE’12. ACM, Sept. 2012. DOI: 10.1145/2351676.2351687 (cit. on p. 18).

[Nyg18] M. Nygard. “Release it!: design and deploy production-ready software”. In: *Release It!* (2018), pp. 1–376 (cit. on p. 4).

[PRS+16] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, D. Vrgoč. “Foundations of JSON Schema”. In: *Proceedings of the 25th International Conference on World Wide Web*. WWW ’16. International World Wide Web Conferences Steering Committee, Apr. 2016. DOI: 10.1145/2872427.2883029 (cit. on p. 7).

[RH09] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empir. Softw. Eng.* 14.2 (2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8 (cit. on p. 35).

[RMBZ21] F. Ramirez, C. Mera-Gomez, R. Bahsoon, Y. Zhang. “An Empirical Study on Microservice Software Development”. In: *2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (SESoS/WDES)*. IEEE, June 2021. DOI: 10.1109/sesos-wdes52566.2021.00008 (cit. on p. 4).

[SBB20] S. Speth, U. Breitenbücher, S. Becker. “Gropius — A Tool for Managing Cross-component Issues”. In: *Software Architecture*. Ed. by H. Muccini, P. Avgeriou, B. Buhnova, J. Camara, M. Caporuscio, M. Franzago, A. Koziolek, P. Scandurra, C. Trubiani, D. Weyns, U. Zdun. Cham: Springer International Publishing, 2020, pp. 82–94. DOI: 10.1007/978-3-030-59155-7_7 (cit. on pp. 5, 7, 9).

[SBB21] S. Speth, S. Becker, U. Breitenbücher. “Cross-Component Issue Metamodel and Modelling Language”. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2021. DOI: 10.5220/0010497703040311 (cit. on p. 5).

[SBK+23] S. Speth, U. Breitenbücher, N. Krieger, P. Wippermann, S. Becker. “Integrating Issue Management Systems of Independently Developed Software Components”. In: *Agile Processes in Software Engineering and Extreme Programming*. Springer Nature Switzerland, 2023, pp. 3–19. DOI: 10.1007/978-3-031-33976-9_1 (cit. on p. 5).

[SBSB23] S. Speth, U. Breitenbücher, S. Stieß, S. Becker. “Dromi: A Tool for Automatically Reporting the Impacts of Sagas Implemented in Microservice Architectures on the Business Processes”. In: *Enterprise Design, Operations, and Computing. EDOC 2022 Workshops*. Ed. by T. P. Sales, H. A. Proper, G. Guizzardi, M. Montali, F. M. Maggi, C. M. Fonseca. Cham: Springer International Publishing, 2023, pp. 326–331. DOI: 10.1007/978-3-031-26886-1_20 (cit. on p. 5).

[SG05] R. J. Sandusky, L. Gasser. “Negotiation and the coordination of information and activity in distributed software problem management”. In: *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work - GROUP ’05*. GROUP ’05. ACM Press, 2005. DOI: 10.1145/1099203.1099238 (cit. on p. 5).

[SGM02] C. Szyperski, D. Gruntz, S. Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002. ISBN: 0201745720 (cit. on p. 4).

[SKBB21] S. Speth, N. Krieger, U. Breitenbücher, S. Becker. “Gropius-VSC: IDE Support for Cross-Component Issue Management”. In: *Companion Proceedings of the 15th European Conference on Software Architecture, CEUR (October 2021)*. 2021 (cit. on p. 6).

[Spe19] S. Speth. “Issue management for multi-project, multi-team microservice architectures”. MA thesis. 2019. DOI: 10.18419/opus-10646 (cit. on pp. 5, 7).

[Spe21] S. Speth. “Semi-automated Cross-Component Issue Management and Impact Analysis”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2021. DOI: 10.1109/ase51524.2021.9678830 (cit. on p. 5).

[SSB22] S. Speth, S. Stieß, S. Becker. “A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis”. In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2022, pp. 116–119. DOI: 10.1109/ICSA-C54293.2022.00029 (cit. on p. 5).

[SSFB22] S. Speth, S. Stieß, S. Frank, S. Becker. “Iterative and incremental refinement of microservice-based architectures and SLOs”. In: (2022). DOI: 10.18419/opus-12208 (cit. on p. 5).

[Thö15] J. Thönes. “Microservices”. In: *IEEE Software* 32.1 (Jan. 2015), pp. 116–116. DOI: 10.1109/ms.2015.11 (cit. on p. 4).

[TLPJ17] D. Taibi, V. Lenarduzzi, C. Pahl, A. Janes. “Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages”. In: *Proceedings of the XP2017 Scientific Workshops*. XP ’17 Workshops. ACM, May 2017. DOI: 10.1145/3120459.3120483 (cit. on p. 4).

[WTEK20] L. Wu, J. Tordsson, E. Elmroth, O. Kao. “MicroRCA: Root Cause Localization of Performance Issues in Microservices”. In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Apr. 2020. DOI: 10.1109/noms47738.2020.9110353 (cit. on p. 13).

All links were last followed on December 19, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature