Lix infrastructure guide

Information about adminstering Lix's infrastructure.

- Infrastructure overview
- Auth/SSO systems
  - Changing names, emails, etc
  - How accounts work
  - How do permissions work?
  - Assigning Groups
  - Tutorial: adding auto mapping of forgejo groups
- Why
- Obliterating history from Git
- Working with S3
- Creating Matrix Rooms/Spaces
- Merging Gerrit identities
- Archived pages
  - Forgejo improvements

Infrastructure overview

The Lix infrastructure is maintained by https://afnix.fr — you can find their public documentation on https://docs.afnix.fr and the infrastructure repository is at https://git.afnix.fr/afnix/infra.

Auth/SSO systems

A major part of Lix infrastructure is the authentication/SSO systems. Here, you can find information about how to run them.

Changing names, emails, etc

The Lix project endeavours to not deadname people, because we believe in human decency. However, some of our software has other ideas. This page documents the workarounds to manually fix profile updates that don't get conducted because various software is busted.

Intended design

Ideally, contributors should be able to go to https://identity.lix.systems and change their usernames, display names and emails and relog every service, and then every service will have correct names and emails.

wiki.lix.systems

The wiki does not update emails when they are changed via OIDC. Furthermore, users can't change them themselves. Why do they do this, we will never know; OIDC has persistent UUIDs, they have no reason to do this.

To fix a user's email manually, go to https://wiki.lix.systems/settings/users, and select the user in question and edit them.

The wiki will also not change fullnames automatically, which is also broken, but users can simply change them. It does not seem to use usernames at all.

git.lix.systems

!! Currently this is broken and we cannot change forgejo usernames at all: https://git.lix.systems/lix-project/web-services/issues/93. The workaround here is to clear the username field when changing to a local account.

Forgejo blocks username changes for accounts with external sign-in for no reason. These have to be fixed by an administrator. Go to https://git.lix.systems/admin/users, click the edit icon next to the user in question, then set the Authentication Source to Local, fix the username, then press Update User Account. Next, set the Authentication Source back to Lix.

Users are able to change emails themselves by adding a new email then deleting the old one. They can also be changed by administrators in the same page as above.

OIDC has persistent UUIDs, there is no reason for Forgejo to do this.

Forgejo does not update names or emails from Keycloak after initial login, which is broken as well.

gerrit.lix.systems

Gerrit will break accounts rendering them incapable of logging in if they change username. Changing email and display name works as expected. It appears that Gerrit wants to believe that usernames are not possible to change, which is a skill issue, because they *have* numeric IDs.

Extremely untested scuffed-looking db hacking procedure:

https://wikitech.wikimedia.org/wiki/SRE/LDAP/Renaming_users/Gerrit

pad.lix.systems

We think this one works properly last time we checked. It seems to just replace the profile on each login.

How accounts work

Lix has one source of truth for authentication: Keycloak (identity.lix.systems). Most services are bound to Keycloak for authentication via OAuth2, although it supports SAML as well.

GitHub vs Local accounts

GitHub accounts are used at Lix for two reasons:

- Ease of login and onboarding
- Ban/allow list management

We don't really care if people have the same username or other information on Lix as they have on GitHub. We don't care about whether people have first/last names on our Keycloak or if they are using pseudonyms.

Allow/ban listing

There is an allow-list and a ban list maintained at: https://git.lix.systems/lix-project/access-control (private repo, available only to Lix core team). To add people to a list, use `./add.sh list.txt gh-username`. Once a list change is pushed, it can take up to five minutes for the change to take effect, as this is currently running on a 5m cron job.

In short, the process for adding a user to the ban or allow list is:

1. Make sure you have the latest version of the repo (i.e. `git pull`) and the github `gh` command is installed.
2. Run `./add.sh <relevant-list-file> <github username>`.
3. Commit and push the change.
4. The ACL change will apply automatically within five minutes.

Be warned -- the allow-list method of access control is temporary / established for the beta period.

Our allow/ban listing is done by GitHub ID, using keycloak-allowban-plugin, a custom Keycloak plugin that reads text files with allow/ban lists. The GitHub ID is put into a user profile attribute, which prevents ban-evasion via account unlinking since it will stick across unlinking.

Known weirdness with the allow/ban list plugin

If a user tries to log in via GitHub and they are not allowed by the plugin, the account is created anyway, it is simply not usable. This is a known issue; putting the plugin in the registration flow

caused half-registered users, so it is only in the post-gh-login flow and the normal login flow (to catch unlinked banned accounts).

Local accounts

The Lix core team should have local accounts (linking to GitHub is OK), strongly preferably with 2FA. Other people can be given local accounts if they are trustworthy and prefer to have local accounts (since the usual ban process doesn't work on them; though it is not hard to ban them, just disable the account).

Note that GitHub backed accounts can be turned *mostly* into local accounts by the user simply setting up local auth and unlinking the GitHub account (though the GitHub ID will intentionally persist in properties so this doesn't degrade our bans story).

We would *prefer* for everyone to use WebAuthn for local accounts, but this is often not possible and passwords are OK as long as they're just put in a password manager.

To create a local account, get the following info:

- Username
- Email (which will appear publicly on Gerrit and must be deliverable)
- WebAuthn ok?

Then create an account on https://identity.lix.systems/admin with the provided details. On the account's page, go to Credentials, select Credential Reset, then if WebAuthn is ok for the person, set "WebAuthn Register Passwordless" in the actions (otherwise just password reset) and send it.

Removing last names for people

Due to Keycloak being a silly little thing, we need to use "declarative user profiles" to allow not setting last names. For now, Lix core team members with necessary access will have to remove them manually on request.

This would be fixed by updating Keycloak to 24 on lix.systems and setting up declarative user profiles: https://git.lix.systems/lix-project/web-services/issues/64

How to ban someone

If a user has violated our community norms and needs to have their access to our infrastructure removed, follow the following steps:

1. Add them to the banned users list on https://git.lix.systems/lix-project/access-control and push the changes.

2. Go to https://identity.lix.systems/admin and disable their account for good measure.
3. Ban them from Matrix: FIXME
4. (if you really don't like what's going on) invalidate all sessions:
    1. `ssh root@git.lix.systems -- mysql -D forgejo -e 'delete from session;'`
    2. `ssh -p 2022 youruser@gerrit.lix.systems gerrit flush-caches --cache web_sessions`
    3. FIXME: bookstack

How do permissions work?

In an ideal world, all permissions are managed directly in Keycloak and propagated down to downstream systems automatically. We mostly live in that world. We would also like more parts of profiles to propagate from Keycloak into downstream systems (see changing names document).

First, let's enumerate the access that we *have available* to grant.

Available access

Roles that exist in Keycloak

"sticky" is referring to whether later-removed permissions get stuck in the downstream system if they are removed upstream

- Wiki roles (not sticky)
  - Editor
  - ModerationBook
  - Admin
- Buildbot (not sticky)
  - can-perform-mutations (currently just a gate on all access)
- Forgejo (administrator is probably sticky, however others are not)
  - Administrator permissions
  - lix-project/nixos issue triage team
  - lix-project owners team
- Keycloak (not sticky)
  - lix-project-user-admin realm role grants administration on keycloak
- Grafana (?)
  - Admin
  - Editor
- Pad (not sticky)
  - access
- Mattermost (not sticky?)
  - access

Roles that we wish existed in Keycloak

These can't happen due to current technical limitations.

- Gerrit: https://git.lix.systems/lix-project/web-services/issues/100
  - Lix team
  - Possibly other rights like tvix-fork team or others?

Structure of access

Keycloak appears to want its structure to work like:

Group -> Composite Realm Role -> Client Role

We don't have that many groups or client roles to assign to make composite realm roles make much sense at the time of this writing.

When creating new clients, make their roles *client roles*, which we can then assign to other objects inside Keycloak so we can do role-based access control at a later time without having to mess with the services.

Groups

- lix core team -> grants administrative permission across a lot of infrastructure:
  - Wiki
    - ModerationBook
    - Admin
    - Editor
  - Forgejo
    - Administrator
    - lix-project owners team
  - Grafana
    - Editor
    - Admin
  - Mattermost
    - access
  - Pad
    - access
  - Buildbot
    - can-perform-mutations
- Lix community team -> grants administrative permission across a lot of infrastructure
  - Wiki
    - ModerationBook
    - Editor
  - Forgejo
    - Administrator
    - community-team
  - Mattermost TODO should probably get access, but currently doesn't (?)

    - Pad
        - access
- trusted-contributors
    - Forgejo
        - lix-project/nixos issue triage team
    - Wiki
        - Editor
    - Buildbot
        - can-perform-mutations

Policy on who goes in groups

- lix core team: *only* members of the Lix core team (this group will probably be rearranged later on)
- lix community team: *only* members of the Lix community team.
- trusted-contributors: anyone who is reasonably competent and trustworthy: these are basically the rights we hand to every single package maintainer in nixpkgs, and we want to do similarly in Lix.
  Especially assign this if someone should be able to have issues assigned to them (since it is a prerequisite); this means this role should be handed to anyone who is going to contribute code so we can use the assignment feature of Forgejo to say who is taking an issue.
    - /private beta: this group has been made to indicate who was in trusted-contributors already due to our simply dumping everyone from the private beta into it

Assigning Groups

See How do permissions work? for implementation details.

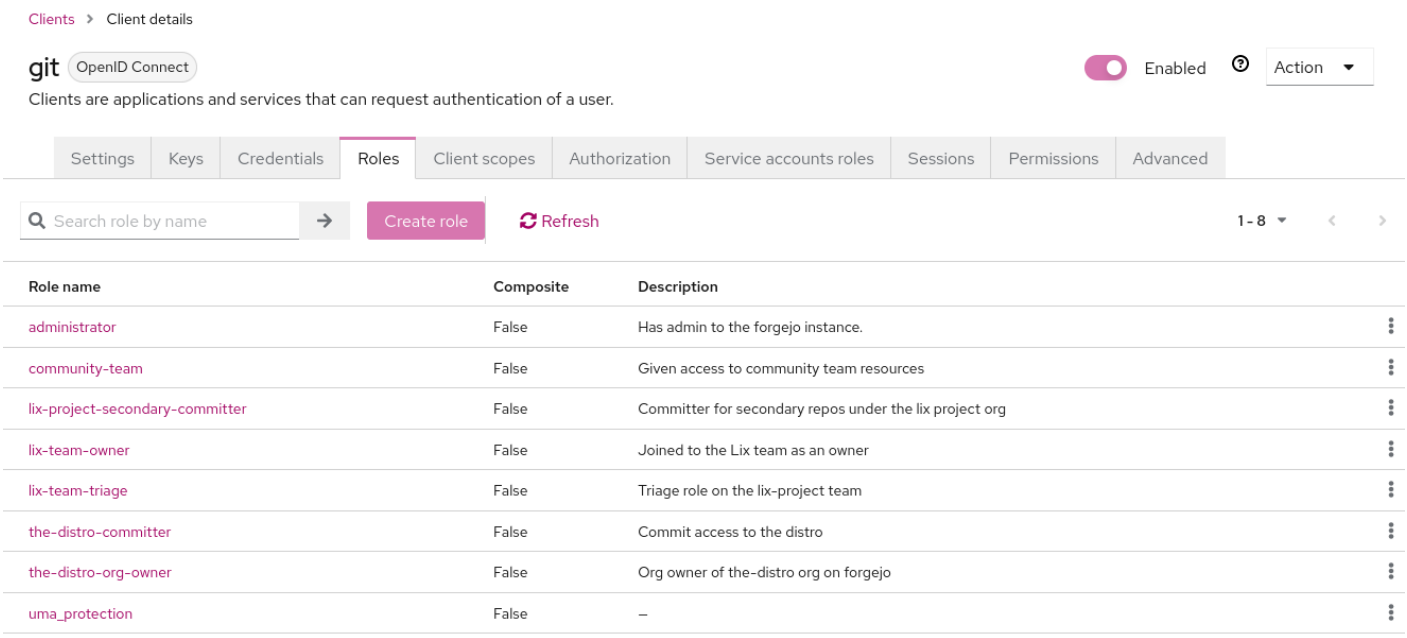
tldr;

- Go to the admin console (no trailing slash)
- Go to Groups -> YourGroupHere -> Members
- Add/Remove members as needed

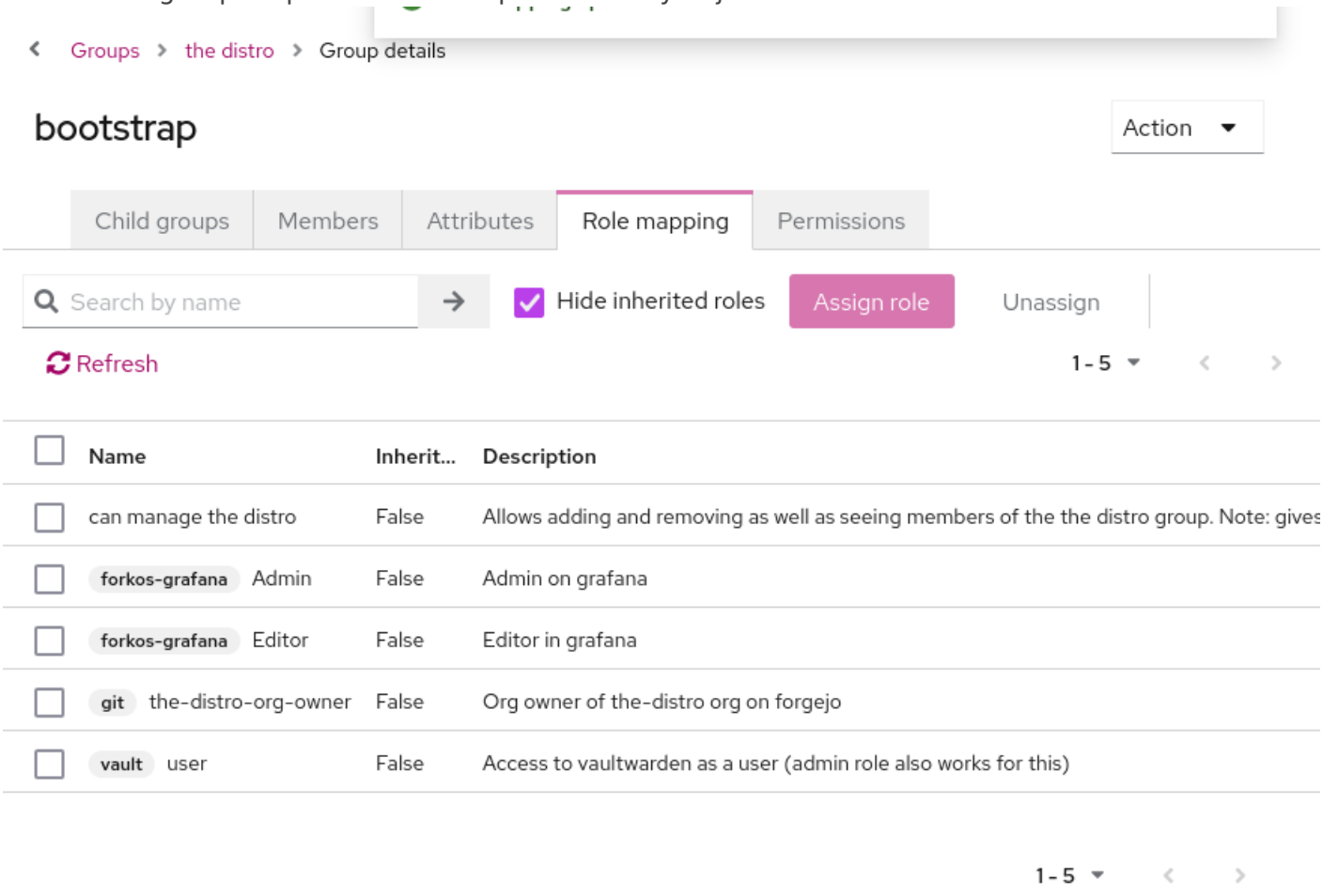
Note: most permissions only update after logging out and back into the appropriate application.

Tutorial: adding auto mapping of forgejo groups

Create a role on the Keycloak client:



Go into the group in question and map it the role you just made:



Add a json snippet to map the role in the incoming tokens to the appropriate team on the org:

```
{"the-distro-committer": {"the-distro": ["committers"]}, "the-distro-org-owner": {"the-distro": ["owners"]}}
```

It needs to be added to: https://git.lix.systems/admin/auths/1

Why

Why? Why self-host all your own infrastructure?

We tried not to, at the very beginning of the project. We agreed that Github-style code review wasn't really fit for our kind of project, and wanted to use Gerrit for code review. We started setting up Gerrithub for a repo hosted on Github, and we ran into so many problems with that approach that it was actually easier to just self-host Gerrit instead (for starters, some members could not log in at all).

Then there was little reason to use GitHub, since none of us are really happy with Github direction lately anyway, and Forgejo + GitHub-enabled SSO mean that contributors shouldn't have to jump through too many hoops to help out.

So now we have a fully independent and open source infrastructure stack, with (hopefully) a good onboarding path as well. And we're also in our own critical path: we run into Nix's papercuts and gashes alike every day, so we better fix it!

Here is our thought process from back then:

- GitHub?
  - Well-known
  - Easy to contribute to as everyone has an account
  - CI would need extra work due to VM tests, e.g.
    - Not a Nix CI system by itself, will be doing self-hosted engineering work for something extra anyway
  - Rubbish code review
  - Can use GerritHub for better review?
    - CI though?
    - Evaluated: found that login does not work and vendor is unresponsive to emails
    - Possible moderation difficulties
- Gitlab/Codeberg?
  - Basically GitHub but by different people
  - Same CI problems as GitHub but fewer off-the-shelf solutions to them
  - Rubbish code review
    - Cannot use GerritHub even assuming that it worked
    - Need self-hosted Gerrit, so need self-hosted Gerrit *auth* and well...
- (self-hosted) forgejo/gitlab
  - Much better control over integrations
  - Same poor code review situation as cloud Forgejo/GitLab

    - Might as well stand up a forgejo if we already have self-hosted Gerrit and the auth for it
- (self-hosted) Gerrit?
    - Easy to integrate with and add plugins to vs GerritHub (e.g. Gerrit does not come with Nix or Meson syntax highlighting)
    - Full control: can fix issues like moderation
    - Works at all
    - Good code review

The Lix CI is broken though!

Yes, our buildbot is a high maintenance service and it *is* janky. Multiple members of the Lix team have plans about writing entirely new Nix CI systems, but they are otherwise busy with another major project in the form of Lix. This is the matrix of extant alternatives:

- buildbot
    - buildbot-nix exists and we have hacked it to use nix-eval-jobs and speak Gerrit with (limited) caveats
        - Logs aren't intermixed
        - Our Gerrit integration is considerably janky auth-wise
        - There *are* bugs
        - UI makes it very non-discoverable which CL caused a build
    - The old Angular UI is mildly busted, and the new React UI has severe accessibility problems and is more busted
    - Knows how to speak Gerrit
- Hydra
    - Due to architectural flaws, it cannot deliver notifications of status to a code review system
        - This is a non-starter
        - Cannot be taught to speak Gerrit because it cannot speak to code review systems
    - Notoriously dubious code and DB schema
    - Logs aren't intermixed
    - Dubious UI
    - Unmaintained for non-hydra.nixos.org use cases (and hydra.nixos.org would rather not use it, but the infra team does not have the cycles to rewrite Hydra)
- Something GitHub Actions based
    - Reinstalls Nix every job
    - Needs a separate binary cache
    - Cloud runners are slow and can't run VM tests, need self-hosted runners
    - Need to solve `nix-eval-jobs` shaped problems
    - All the logs go into one stream, rather than being per-derivation, so error reporting is challenged
    - Tied to GitHub, impractical to attach to Gerrit

- Garnix
  - Tied to GitHub, impractical to attach to Gerrit
  - Unclear if it can run VM tests
  - Closed source, not observable
- Woodpecker
  - Build your own Nix CI!
  - FIXME: add more about this

Obliterating history from Git

To obliterate history from the Git repo means removing it from three different sources: Gerrit, Forgejo, and GitHub.

A tool has been written, called gerrit-rewrite-branch, to rewrite Gerrit history completely, including the meta on past CLs.

To use it, build it as `--release` (it will stack overflow on debug mode), and find the following repos, and make backups of them:

- `/var/lib/gerrit/git/lix.git`
- `/var/lib/forgejo/repositories/*/lix.git`

To start off, stop Gerrit and find the Git repo for it. The tool requires four things: The email address to obliterate, and a replacement name + email address. It also needs a cutoff date for where to remove commits before. To find this, run `git cat-file -p {commit}` for a commit earlier than the oldest you want to remove, and note down the timestamp on the `committer` line.

Call the tool. It will churn for a while, and rewrite all previous Git commits, plus the Gerrit metadata of affected commits. As a bonus, run a `git gc --prune=now`.

Before turning on Gerrit, run `systemd-run -p DynamicUser=yes -p StateDirectory=gerrit -t gerrit reindex -d /var/lib/gerrit`. This ensures Gerrit is aware of the changes made outside of its existence.

For forgejo, no special steps are needed; just run the same tool over these repos plus all their forks, and prune the reflog and unreachable commits as well:

```
[root@lix:/var/lib/forgejo/repositories]# for i in */lix.git; do pushd $i; sudo -u git git reflog expire --expire=all --expire-unreachable=all --all; sudo -u git git gc --prune=now; popd; done
```

Once Gerrit and Forgejo are back up, run `ssh gerrit.lix.systems replication start --now --url github` to propagate the changes to GitHub.

Don't forget to ban the commits as well, using `ssh gerrit.lix.systems gerrit ban-commit lix {commits}`.

Working with S3

Introduction

We use garage, an open-source server compatible with Amazon's S3 API, hosted on our own infrastructure. Currently we store both documentation and binaries there; it may be used for other things down the line.

Configuring a client

You probably want to use rclone; it's friendly (to people who like the terminal) and not tightly bound to any specific storage service. You can also use the Amazon first-party S3 tooling, but this guide does not attempt to explain how.

To follow these steps, you will need to already have ssh access to the server garage runs on, which is `s3.lix.systems`. As of this writing, there is no guide about how to do that, but take a look at `services/ssh.nix` in the `web-services` repo and see whether it makes sense to you. Please feel free to write said guide and add it to this wiki. :)

Generating S3 credentials

Once you have ssh access, you will need to make s3 credentials. You can do it like this:

```
$ ssh root@s3.lix.systems
[root@cache:~]# garage key create some-key-name
Key name: some-key-name
Key ID: GKa653da6819c4140c3db9dfc5
Secret key: ab2b6106fbb7681517cba875c26c8ea99e281f113e2fd809decd6e524ebbc639

Can create buckets: false

Key-specific bucket aliases:

Authorized buckets:
```

(Don't worry - those aren't real keys there, nor were they ever! They're synthetic examples so you know what they look like.)

You'll want to choose a key name that helps the rest of us know whose it is and what it's used for. Don't just create a key called `some-key-name` by copying the example verbatim, it will be confusing clutter!

The most important criterion for a key name is that reading the name should let you answer the questions "is anyone still using this?" and "what will break if this key is deleted?" If you need naming inspiration you can see other people's key names with `garage key list`; in particular, keys meant for individual use should probably start with your username.

Before you sign out of the server, also make sure to grant the key the permissions you need. For example, if you need to work with the `docs` bucket, do:

```
[root@cache:~]# garage bucket allow --read --write docs --key irenes-temp-delete
New permissions for GKa653da6819c4140c3db9dfc5 on docs: read true, write true, owner false.
```

You can see what buckets exist by doing `garage bucket list`.

The "Key ID" and "Secret key" values from the key you generated are what you'll need in the next step. Make sure you have them; there's no way to look up the secret part later.

Configuring your client (probably rclone)

You may find it useful to reference the garage documentation on this.

There are two ways to configure rclone, either of which will work. The one Irenes recommend is to put the credentials directly into the rclone configuration (it has its own tooling for securing them, which you can set up if you want). The other way is to let rclone read them out of the config file used by Amazon's first-party tooling. Either will work; using the AWS config file is a little harder to figure out what you did later, if you happen to forget. Also, if Lix infrastructure isn't the only S3 service you use on a regular basis, the rclone config is probably a better place to keep track of everything because AWS profiles are a pain to use.

If you're doing it the Irenes way, you can either run `rclone config` and go through the prompts, or just prepare a config file by hand.

Here's a sample rclone.config:

```
[lix]
type = s3
provider = Other
env_auth = false
endpoint = s3.lix.systems
region = garage
access_key_id = GKa653da6819c4140c3db9dfc5
```

```
secret_access_key = ab2b6106fbb7681517cba875c26c8ea99e281f113e2fd809decd6e524ebbc639
```

For more information about where to put this config file, see `man rclone`; it's likely that `~/.config/rclone/rclone.conf` is the right place.

Please notice that this example file uses `lix` as the name of the rclone "remote". That means that, when interacting with it, you'll use paths like `lix:` to refer to the entire thing, or `lix:docs/` to refer to the root of the bucket named `docs`, and so on. You can use any name you find convenient for the remote, it doesn't have to be `lix`, but this document will assume it's that. If you think you might have done this configuration already but don't remember what you called the remote, do `rclone listremotes`.

If you're going through the interactive configuration, choose the generic S3-compatible service as the type of service. For the endpoint, write in `s3.lix.systems`, and for the region, write in `garage`. If you leave region blank you'll get weird errors about us-east-1, but we're not Amazon and we don't have a global network of highly-redundant data centers, so don't leave it blank. :)

If you're storing credentials in the AWS config file, everything is pretty similar except you'll need to prepare `~/.aws/credentials` yourself, and tell rclone to use it; the `rclone config` wizard has options for that. The easy way is to use the `default` profile in the AWS credentials file; otherwise you'll have to make sure your environment sets `AWS_PROFILE`, since rclone has no option to manage that itself.

Copying files into and out of s3

If you're using `rclone`, you may find it useful to do `rclone ls lix:` to get a sense of what's there. This will probably become increasingly bad advice as our usage of S3 grows! :)

Notice that the first path component in this output is the bucket name, so ie. a file named `index.html` at the root of the `docs` bucket is listed as `docs/index.html` in this view. That is also how you will refer to it from the command line. Other S3 clients have different conventions in this regard, so if you're using something else, check its upstream documentation.

If you have a local file `index.html` and you want to overwrite the remote `docs/index.html` with it, do `rclone copy index.html lix:docs/`. You have to give a directory prefix, not a filename, for the second part.

In general, the `rclone` CLI lets you intermingle local and remote paths, so pay close attention to the colons. `lix:something` is a remote path, `something` is a local one. If you lose track of this you will end up sad.

For any other `rclone`-related questions, `rclone --help` and `man rclone` are good references.

Happy filing!

Creating Matrix Rooms/Spaces

Matrix room version >= v12 give implicit infinite power level to any creator(s) of the room. When creating rooms, make sure that they are owned (created by) `@admin:synapse.lix.systems` and that account only. Given that most clients don't expose any of such "internals" in the UI yet, the easiest way is to impersonate the admin user via access token or password login.

TODO document where we store the password / how to get an access token.

- Create the room via API or client from the admin user
- Give yourself permissions as necessary

We use https://github.com/tulir/mauliasproxy to forward all our aliases from `lix.systems` to `synapse.lix.systems`. Once a local alias is added, the forwarder will work and the main address can be set:

- Configure a local room alias: `@myroom:synapse.lix.systems`
- Then add a remote alias on lix.systems: `@myroom:lix.systems`
  - Set that as the main address

If using draupnir to protect that room:

- make sure @draupnir:lix.systems is invited to the room and has the maximum available power level in the room.
  - Draupnir can join with `!draupnir rooms add <roomid>`
- Add room to the appropriate spaces. This requires respective permissions in the space's room

Merging Gerrit identities

Basically, following https://ovirt-infra-docs.readthedocs.io/en/latest/General/Gerrit_account_merge/index.html.

If for some reason, you don't have access to `refs/meta/external-ids`, you can still do it on the server directly as long as you ensure that you restore the permbits for `gerrit:gerrit` on the git storage.

You can extract a worktree `git worktree add /tmp/external-ids refs/meta/external-ids`, generate a commit that fixes things and you can complete by `git update-ref refs/meta/external-ids $commit_sha1`.

Note:

- that CLs under dashboard of the previous account will just disappear and won't go to the new account.
- HTTP passwords are generated under the old username (?)
- prefer always to take the account with the most changes/comments.

Archived pages

Those pages contains outdated information but are still relevant from an history perspective to understand the Lix infrastructure.

Forgejo improvements

A brief overview of our code infrastructure for those not in the Lix project:

- Forgejo https://git.lix.systems for issue tracking for Lix and other projects, as well as for project boards. It also provides a read-only git mirror of Lix, forks for WIP code in Lix, and hosts minor projects like infra and other things that use PR workflows.
- Bookstack https://wiki.lix.systems as wiki. It's a more normal wiki program, which has working search among other niceties. Also, it presents the whole wiki as one entity.
- Gerrit https://gerrit.lix.systems for major code review. We have seen the Forgejo gerrit-like patch workflow and it looks great, but the review UX is not at all like Gerrit and makes way different prioritization of space.
  Not being like Gerrit is largely fine for Forgejo's goals, since it is not aiming to be Gerrit-like, it is aiming to be GitHub like (and GitHub has quite bad review UX); we would suggest supporting Change-Ids to have unambiguous change associations, but we probably won't use that feature anyway as we have Gerrit available.
- Buildbot https://buildbot.lix.systems (private link) for CI. We need something quite laser focused on being good at CI'ing Nix expressions and buildbot-nix is generally the least broken one of those.
- Keycloak https://identity.lix.systems for SSO. It is comprehensive across our entire infrastructure, and it is the *only* login system in our infrastructure. We use GitHub as an upstream for anti-spam + moderation reasons and additionally issue a limited number of local accounts on Keycloak.

Stuff that works great

Forgejo does a lot of stuff better than GitHub and we love it very much for these things.

- The web interface is snappy!
- Syncing groups from Keycloak is a breeze.
- We can patch it. Gosh. This helps so much.
- We only rarely run into bugs.
- The devs have generally been quite responsive!
- The release notes are generally comprehensive and cover the issues we run into on upgrades (such as the theme rename and the UTF8 column type thing in mysql).

Stuff that makes us Very Sad

- Forgejo is a major source of sticky user metadata

  https://codeberg.org/forgejo/forgejo/issues/3657 and

  https://codeberg.org/forgejo/forgejo/issues/3682. This requires admin intervention if the username *needs* to be changed, and causes confusion in administering the system in general, since usernames don't match with Keycloak so you have to find users in Keycloak by ID. We want selectable auto-synced user metadata from SSO combined with locking such synced metadata out on the Forgejo side to prevent confusion.

  See also: https://wiki.lix.systems/link/11#bkmrk-git.lix.systems
- A mild sadness, but it is not great: You can't make branch protection rules that forbid all human pushes yet allow force pushes; branch protection always implies no force pushes

  https://codeberg.org/forgejo/forgejo/issues/7238

Stuff that would be Nice

UX

- Top simple request, which we are quite likely to do ourselves: something like Gerrit commentlinks: https://gerrit-review.googlesource.com/Documentation/config-gerrit.html#commentlink
  That is, arbitrary regexes that can inject links into the page, which are applied in comments and in commit messages. This use case is not served for us by the external issue tracker feature of Forgejo, since we want to link stuff like `Change-Id: I1360750d4181ce1ca2a3aa4dc0e97e131351c469`, which is *not* an external issue, it is external code review. Also, we want to have more than one of them, for things like `cl/123` which should go to the Gerrit changelist 123.
  Gerrit has these either installation or repo scoped, and we use both on our installation: Change-Id and cl/ gets linked everywhere since it is global, and issue refs like `#123` get linked just on the Lix project since it is project specific.
  Also, I think that the external issue tracker feature might not work well for its intended use case anyhow? On GitHub you can have autolinks of TEAMONE-123 and TEAMTWO-123 and I don't know if Forgejo knows how to do that?

  Hacked in with an evil local patch: https://gist.github.com/lf-/f2e31a329c3c48f09198c865e21618e6
- Ability to put arbitrary links at the top of repos next to Issues, etc.
  We want this for linking to our Gerrit reviews, so it shouldn't be called Pull Requests, but idk there might be another way to do this that's better.

  Hacked in with an evil local patch: https://gist.github.com/lf-/f2e31a329c3c48f09198c865e21618e6
- jade's personal complaint: I can't hit `t` to fuzzy-find files like I can on GitHub!
- jade's other personal complaint: issue comment fields aren't saved across page changes, and they *also* can get desynced such that the comment was posted but you get a warning on navigating off of the page.

- Issue categories display differently if you have a / in them if they are exclusive or not. This is probably on purpose, but it does make our issue labels look a little bit funny when they *are* all hierarchical, just not all of the categories are exclusive.
- We would like to be able to replace the explore page with the Lix organization page, since we don't really want people enumerating our users, and it is the only thing that really matters significantly on our instance.

Operations

- Easier management of bot accounts, perhaps by attaching them to an org or something; we would like to issue tokens for bot use cases at an org level rather than having to make weird local accounts and put them in a password manager.
- Git operations are strangely slow on our instance, taking about 5 seconds to push on seemingly every repo. We aren't 100% sure why this is, and don't know how to profile it. The web interface is snappy.
- We have a hacky prototype to use SCIP code databases to provide perfect go-to-definition in the code browser, but we haven't operationalized it with CI builds etc. It would be nice if Forgejo knew how to handle this natively.
  - puck> A big issue with this is that this would preferably hook into the syntax highlighter, to ensure spans don't pass SCIP token boundaries. The prototype I put together does some trickery to try and figure out exactly which character you clicked on. This means it works terribly on mobile due to click target sizes. (And it's missing a nice popup.)
- You can't move issues between repos.
- Repo-scoped project boards are somewhat useless since you can't include issues from multiple repos. I am not sure if you can move repo-scoped boards to org-scoped either.
- It isn't possible to pause the mail queue by setting workers to 0. Or to flush the mail queue when the mailer is disabled.
  The reason we are writing this is that there was such an operational incident where we were about to send 900 emails that nobody wanted because of an issue-copying script (which later had a dont_notify API param added to not try to send the emails in the first place). I think we deleted the queue directory or broke the mail config on purpose and then hit the flush button or something.
  We have worked around this ourselves but we wanted to mention it.
- We have noticed that moderation tools are somewhat lacking in Forgejo compared to GitHub. For example you can't lock non-member participation in issues if some heated external event happens (but we would probably just hack our Keycloak allow/ban plugin to reject registrations).
  - Although it's not something we need as we can do it with SSO+dumping the session cache, I don't think you can ban people on Forgejo?
  - We would like to be able to time people out
- We can't easily redact/replace commit IDs from the activity log since they are stored as HTML. The purpose of this is to be able to perform Git history rewriting in case we want to wipe someone's deadname from history and fix it. This is not urgent, as in, we don't know when we will need this, but it will plausibly happen one day; I think we were considering

writing tooling to fix it up in the DB though.
    - puck> The event history of a repository will keep old commits IDs, and their summaries, even if the commit no longer exists.
    - puck> The tooling to rewrite commit IDs has been written, but is a massive hack (see how to use it): it takes in a `mysqldump` and rewrites any mention of the commit ID. It works, though ^_^
    - puck> This might be a big ask, but it'd be great to have an equivalent to Gerrit's "banned commits" - commit hashes that, if encountered on any push, reject the push instantly.
- Teams don't have a check box to auto-add them to new public repos. We use teams to grant wide reaching issue triage permissions on everything (this works *fantastically* for us as a community, and nixpkgs also does this), but we have some private repos we don't wish to grant access on, so we had to set the team to not all repos.
- There are no org-scoped milestones, which is somewhat annoying as we are using forgejo for *planning*-related issues that aren't in the Lix repo itself, but do block release. We have used projects for now and it's generally fine.
- It isn't possible to continuously mirror new issues on a repo from GitHub.
    - Relatedly, it is not possible to *limit creating new issues on a repo* and thus messing up the numbering of such a mirror
    - We don't need this that bad since we seem to be going on a very-hard fork course of not worrying about the upstream issue tracker anymore.

Stuff we patched that could probably be done Better upstream

- We added a Nix tarball link feature, which is now upstream! □□
- In routers/web/auth/auth.go we have patched a redirect in to just unconditionally go to OAuth2 when you hit the login button. This is kind of a hack, and it prevents logging in with local accounts, which we have an extremely limited use of for a bot account. Probably the good implementation here is this but with a `?local` url parameter or so.
- We have an issue mirror of nixos/nix on GitHub (which is likely to be deprecated soon, to be fair! so don't worry too much about it) and we did this to add a button to get to upstream issues; in general adding navigation in the tool would be good:

```
diff --git a/templates/repo/issue/view_title.tmpl
b/templates/repo/issue/view_title.tmpl
index c1dd265..9a573ce 100644
--- a/templates/repo/issue/view_title.tmpl
+++ b/templates/repo/issue/view_title.tmpl
@@ -16,6 +16,9 @@
 □□□{{if and (or .HasIssuesOrPullsWritePermission .IsIssuePoster) (not
.Repository.IsArchived)}}
```

```
 □□□□<button id="edit-title" class="ui small basic button edit-button not-in-edit{{if
.Issue.IsPull}} gt-mr-0{{end}}">{{.locale.Tr "repo.issues.edit"}}</button>
 □□□{{end}}
+□□□{{if and (eq .Repository.Name "nix") (eq .Repository.OwnerName "NixOS")}}
+□□□□<a role="button" class="ui small basic button"
href="https://github.com/NixOS/nix/issues/{{.Issue.Index}}">{{svg "octicon-mark-
github"}} Open upstream</a>
+□□□{{end}}
 □□□{{if not .Issue.IsPull}}
 □□□□<a role="button" class="ui small green button new-issue-button gt-mr-0"
href="{{.RepoLink}}/issues/new{{if
.NewIssueChooseTemplate}}/choose{{end}}">{{.locale.Tr "repo.issues.new"}}</a>
 □□□{{end}}
```

- We have an extremely scuffed patch to add a `dont_notify` parameter on the issue creation endpoint. This is so you can build automation that creates a giant pile of issues that you want in the tracker but which don't need to notify anyone.