

Introduction

Welcome to the documentation repository for Synapse, a **Matrix** homeserver implementation developed by Element.

Installing and using Synapse

This documentation covers topics for **installation**, **configuration** and **maintenance** of your Synapse process:

- Learn how to [install](#) and [configure](#) your own instance, perhaps with [Single Sign-On](#).
- See how to [upgrade](#) between Synapse versions.
- Administer your instance using the [Admin API](#), installing [pluggable modules](#), or by accessing the [manhole](#).
- Learn how to [read log lines](#), [configure logging](#) or set up [structured logging](#).
- Scale Synapse through additional [worker processes](#).
- Set up [monitoring and metrics](#) to keep an eye on your Synapse instance's performance.

Developing on Synapse

Contributions are welcome! Synapse is primarily written in [Python](#). As a developer, you may be interested in the following documentation:

- Read the [Contributing Guide](#). It is meant to walk new contributors through the process of developing and submitting a change to the Synapse codebase (which is [hosted on GitHub](#)).
- Set up your [development environment](#), then learn how to [lint](#) and [test](#) your code.
- Look at [the issue tracker](#) for bugs to fix or features to add. If you're new, it may be best to start with those labeled [good first issue](#).
- Understand [how Synapse is built](#), how to [migrate database schemas](#), learn about [federation](#) and how to [set up a local federation](#) for development.
- We like to keep our [git](#) history clean. [Learn](#) how to do so!

- And finally, contribute to this documentation! The source for which is [located here](#).

Reporting a security vulnerability

If you've found a security issue in Synapse or any other Element project, please report it to us in accordance with our [Security Disclosure Policy](#). Thank you!

Installation Instructions

Choosing your server name

It is important to choose the name for your server before you install Synapse, because it cannot be changed later.

The server name determines the "domain" part of user-ids for users on your server: these will all be of the format `@user:my.domain.name`. It also determines how other matrix servers will reach yours for federation.

For a test configuration, set this to the hostname of your server. For a more production-ready setup, you will probably want to specify your domain (`example.com`) rather than a matrix-specific hostname here (in the same way that your email address is probably `user@example.com` rather than `user@email.example.com`) - but doing so may require more advanced setup: see [Setting up Federation](#).

Installing Synapse

Prebuilt packages

Prebuilt packages are available for a number of platforms. These are recommended for most users.

Docker images and Ansible playbooks

There is an official synapse image available at <https://hub.docker.com/r/matrixdotorg/synapse> or at ghcr.io/element-hq/synapse which can be used with the docker-compose file available at [contrib/docker](#). Further information on this including configuration options is available in the README on hub.docker.com.

Alternatively, Andreas Peters (previously Silvio Fricke) has contributed a Dockerfile to automate a synapse server in a single Docker image, at <https://hub.docker.com/r/avhost/docker-matrix/tags/>

Slavi Pantaleev has created an Ansible playbook, which installs the official Docker image of Matrix Synapse along with many other Matrix-related services (Postgres database, Element, coturn, ma1sd, SSL support, etc.). For more details, see <https://github.com/spantaleev/matrix-docker-ansible-deploy>

Debian/Ubuntu

Matrix.org packages

Matrix.org provides Debian/Ubuntu packages of Synapse, for the amd64 architecture via <https://packages.matrix.org/debian/>.

To install the latest release:

```
sudo apt install -y lsb-release wget apt-transport-https
sudo wget -O /usr/share/keyrings/matrix-org-archive-keyring.gpg
https://packages.matrix.org/debian/matrix-org-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/matrix-org-archive-keyring.gpg]
https://packages.matrix.org/debian/ $(lsb_release -cs) main" |
  sudo tee /etc/apt/sources.list.d/matrix-org.list
sudo apt update
sudo apt install matrix-synapse-py3
```

Packages are also published for release candidates. To enable the prerelease channel, add `prerelease` to the `sources.list` line. For example:

```
sudo wget -O /usr/share/keyrings/matrix-org-archive-keyring.gpg
https://packages.matrix.org/debian/matrix-org-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/matrix-org-archive-keyring.gpg]
https://packages.matrix.org/debian/ $(lsb_release -cs) main prerelease" |
  sudo tee /etc/apt/sources.list.d/matrix-org.list
sudo apt update
sudo apt install matrix-synapse-py3
```

The fingerprint of the repository signing key (as shown by `gpg /usr/share/keyrings/matrix-org-archive-keyring.gpg`) is `AAF9AE843A7584B5A3E4CD2BCF45A512DE2DA058`.

When installing with Debian packages, you might prefer to place files in `/etc/matrix-synapse/conf.d/` to override your configuration without editing the main configuration file at `/etc/matrix-synapse/homeserver.yaml`. By doing that, you won't be asked if you want to replace your configuration file when you upgrade the Debian package to a later version.

Downstream Debian packages

Andrej Shadura maintains a `matrix-synapse` package in the Debian repositories. For `bookworm` and `sid`, it can be installed simply with:

```
sudo apt install matrix-synapse
```

Synapse is also available in `bullseye-backports`. Please see the [Debian documentation](#) for information on how to use backports.

`matrix-synapse` is no longer maintained for `buster` and older.

Downstream Ubuntu packages

We do not recommend using the packages in the default Ubuntu repository at this time, as they are [old and suffer from known security vulnerabilities](#). The latest version of Synapse can be installed from [our repository](#).

Fedora

Synapse is in the Fedora repositories as [matrix-synapse](#):

```
sudo dnf install matrix-synapse
```

Additionally, Oleg Girko provides Fedora RPMs at
<https://obs.infoserver.lv/project/monitor/matrix-synapse>

OpenSUSE

Synapse is in the OpenSUSE repositories as [matrix-synapse](#):

```
sudo zypper install matrix-synapse
```

SUSE Linux Enterprise Server

Unofficial package are built for SLES 15 in the openSUSE:Backports:SLE-15 repository at
<https://download.opensuse.org/repositories/openSUSE:/Backports:/SLE-15/standard/>

ArchLinux

The quickest way to get up and running with ArchLinux is probably with the package provided by ArchLinux https://archlinux.org/packages/extra/x86_64/matrix-synapse/, which should pull in most of the necessary dependencies.

pip may be outdated (6.0.7-1 and needs to be upgraded to 6.0.8-1):

```
sudo pip install --upgrade pip
```

If you encounter an error with lib bcrypt causing an Wrong ELF Class: ELFCLASS32 (x64 Systems), you may need to reinstall py-bcrypt to correctly compile it under the right architecture. (This should not be needed if installing under virtualenv):

```
sudo pip uninstall py-bcrypt
sudo pip install py-bcrypt
```

Alpine Linux

Jahway603 maintains [Synapse packages for Alpine Linux](#) in the community repository. Install with:

```
sudo apk add synapse
```

Void Linux

Synapse can be found in the void repositories as '[synapse](#)':

```
xbps-install -S  
xbps-install -S synapse
```

FreeBSD

Synapse can be installed via FreeBSD Ports or Packages contributed by Brendan Molloy from:

- Ports: `cd /usr/ports/net-im/py-matrix-synapse && make install clean`
- Packages: `pkg install py38-matrix-synapse`

OpenBSD

As of OpenBSD 6.7 Synapse is available as a pre-compiled binary. The filesystem underlying the homeserver directory (defaults to `/var/synapse`) has to be mounted with `wxallowed` (cf. `mount(8)`), so creating a separate filesystem and mounting it to `/var/synapse` should be taken into consideration.

Installing Synapse:

```
doas pkg_add synapse
```

NixOS

Robin Lambertz has packaged Synapse for NixOS at:

<https://github.com/NixOS/nixpkgs/blob/master/nixos/modules/services/matrix/synapse.nix>

Installing as a Python module from PyPI

It's also possible to install Synapse as a Python module from PyPI.

When following this route please make sure that the [Platform-specific prerequisites](#) are already installed.

System requirements:

- POSIX-compliant system (tested on Linux & OS X)
- Python 3.9 or later, up to Python 3.13.
- At least 1GB of free RAM if you want to join large public rooms like #matrix:matrix.org

If building on an uncommon architecture for which pre-built wheels are unavailable, you will need to have a recent Rust compiler installed. The easiest way of installing the latest version is to use [rustup](#).

To install the Synapse homeserver run:

```
mkdir -p ~/synapse
virtualenv -p python3 ~/synapse/env
source ~/synapse/env/bin/activate
pip install --upgrade pip
pip install --upgrade setuptools
pip install matrix-synapse
```

This will download Synapse from [PyPI](#) and install it, along with the python libraries it uses, into a virtual environment under `~/synapse/env`. Feel free to pick a different directory if you prefer.

This Synapse installation can then be later upgraded by using pip again with the update flag:

```
source ~/synapse/env/bin/activate
pip install -U matrix-synapse
```

Before you can start Synapse, you will need to generate a configuration file. To do this, run (in your virtualenv, as before):

```
cd ~/synapse
python -m synapse.app.homeserver \
--server-name my.domain.name \
--config-path homeserver.yaml \
--generate-config \
--report-stats=[yes|no]
```

... substituting an appropriate value for `--server-name` and choosing whether or not to report usage statistics (hostname, Synapse version, uptime, total users, etc.) to the developers via the `--report-stats` argument.

This command will generate you a config file that you can then customise, but it will also generate a set of keys for you. These keys will allow your homeserver to identify itself to other homeservers, so don't lose or delete them. It would be wise to back them up somewhere safe. (If, for whatever reason, you do need to change your homeserver's keys, you may find that other homeservers have the old key cached. If you update the signing key, you should change the name of the key in the `<server name>.signing.key` file (the second word) to something different. See the [spec](#) for more information on key management).

To actually run your new homeserver, pick a working directory for Synapse to run (e.g. `~/synapse`), and:

```
cd ~/synapse
source env/bin/activate
synctl start
```

Platform-specific prerequisites

Synapse is written in Python but some of the libraries it uses are written in C. So before we can install Synapse itself we need a working C compiler and the header files for Python C extensions.

Debian/Ubuntu/Raspbian

Installing prerequisites on Ubuntu or Debian:

```
sudo apt install build-essential python3-dev libffi-dev \
    python3-pip python3-setuptools sqlite3 \
    libssl-dev virtualenv libjpeg-dev libxslt1-dev libicu-dev
```

ArchLinux

Installing prerequisites on ArchLinux:

```
sudo pacman -S base-devel python python-pip \
    python-setuptools python-virtualenv sqlite3 icu
```

CentOS/Fedora

Installing prerequisites on CentOS or Fedora Linux:

```
sudo dnf install libtiff-devel libjpeg-devel libzip-devel freetype-devel \
    libwebp-devel libxml2-devel libxslt-devel libpq-devel \
    python3-virtualenv libffi-devel openssl-devel python3-devel \
    libicu-devel
sudo dnf group install "Development Tools"
```

Red Hat Enterprise Linux / Rocky Linux / Oracle Linux

Note: The term "RHEL" below refers to Red Hat Enterprise Linux, Oracle Linux and Rocky Linux. The distributions are 1:1 binary compatible.

It's recommended to use the latest Python versions.

RHEL 8 in particular ships with Python 3.6 by default which is EOL and therefore no longer supported by Synapse. RHEL 9 ships with Python 3.9 which is still supported by the Python

core team as of this writing. However, newer Python versions provide significant performance improvements and they're available in official distributions' repositories. Therefore it's recommended to use them.

Python 3.11 and 3.12 are available for both RHEL 8 and 9.

These commands should be run as root user.

Install new version of Python. You only need one of these:

```
# Python 3.11
dnf install python3.11 python3.11-devel
```

```
# Python 3.12
dnf install python3.12 python3.12-devel
```

Finally, install common prerequisites

```
dnf install libicu libicu-devel libpq5 libpq5-devel lz4 pkgconf
dnf group install "Development Tools"
```

Using venv module instead of virtualenv command

It's recommended to use Python venv module directly rather than the virtualenv command.

- On RHEL 9, virtualenv is only available on [EPEL](#).
- On RHEL 8, virtualenv is based on Python 3.6. It does not support creating 3.11/3.12 virtual environments.

Here's an example of creating Python 3.12 virtual environment and installing Synapse from PyPI.

```
mkdir -p ~/synapse
# To use Python 3.11, simply use the command "python3.11" instead.
python3.12 -m venv ~/synapse/env
source ~/synapse/env/bin/activate
pip install --upgrade pip
pip install --upgrade setuptools
pip install matrix-synapse
```

macOS

Installing prerequisites on macOS:

You may need to install the latest Xcode developer tools:

```
xcode-select --install
```

Some extra dependencies may be needed. You can use Homebrew (<https://brew.sh>) for them.

You may need to install icu, and make the icu binaries and libraries accessible. Please follow [the official instructions of PyICU](#) to do so.

If you're struggling to get icu discovered, and see:

`RuntimeError:`

`Please install pkg-config on your system or set the ICU_VERSION environment variable to the version of ICU you have installed.`

despite it being installed and having your `PATH` updated, you can omit this dependency by not specifying `--extras all` to `poetry`. If using postgres, you can install Synapse via `poetry install --extras saml2 --extras oidc --extras postgres --extras opentracing --extras redis --extras sentry`. ICU is not a hard dependency on getting a working installation.

On ARM-based Macs you may also need to install libjpeg and libpq:

```
brew install jpeg libpq
```

On macOS Catalina (10.15) you may need to explicitly install OpenSSL via brew and inform `pip` about it so that `psycopg2` builds:

```
brew install openssl@1.1
export LDFLAGS="-L/usr/local/opt/openssl/lib"
export CPPFLAGS="-I/usr/local/opt/openssl/include"
```

OpenSUSE

Installing prerequisites on openSUSE:

```
sudo zypper in -t pattern devel_basis
sudo zypper in python-pip python-setuptools sqlite3 python-virtualenv \
    python-devel libffi-devel libopenssl-devel libjpeg62-devel \
    libicu-devel
```

OpenBSD

A port of Synapse is available under `net/synapse`. The filesystem underlying the homeserver directory (defaults to `/var/synapse`) has to be mounted with `wxallowed` (cf. `mount(8)`), so creating a separate filesystem and mounting it to `/var/synapse` should be taken into consideration.

To be able to build Synapse's dependency on python the `WRKOBJDIR` (cf. [bsd.port.mk\(5\)](#)) for building python, too, needs to be on a filesystem mounted with `wxallowed` (cf. `mount(8)`).

Creating a `WRKOBJDIR` for building python under `/usr/local` (which on a default OpenBSD installation is mounted with `wxallowed`):

```
doas mkdir /usr/local/pobj_wxallowed
```

Assuming `PORTS_PRIVSEP=Yes` (cf. `bsd.port.mk(5)`) and `SUDO=doas` are configured in `/etc/mk.conf`:

```
doas chown _pbuild:_pbuild /usr/local/pobj_wxallowed
```

Setting the `WRKOBJDIR` for building python:

```
echo WRKOBJDIR_lang/python/3.7=/usr/local/pobj_wxallowed
\\nWRKOBJDIR_lang/python/2.7=/usr/local/pobj_wxallowed >> /etc/mk.conf
```

Building Synapse:

```
cd /usr/ports/net/synapse
make install
```

Windows

Running Synapse natively on Windows is not officially supported.

If you wish to run or develop Synapse on Windows, the Windows Subsystem for Linux provides a Linux environment which is capable of using the Debian, Fedora, or source installation methods. More information about WSL can be found at <https://docs.microsoft.com/en-us/windows/wsl/install> for Windows 10/11 and <https://docs.microsoft.com/en-us/windows/wsl/install-on-server> for Windows Server.

Setting up Synapse

Once you have installed synapse as above, you will need to configure it.

Using PostgreSQL

By default Synapse uses an `SQLite` database and in doing so trades performance for convenience. Almost all installations should opt to use `PostgreSQL` instead. Advantages include:

- significant performance improvements due to the superior threading and caching model, smarter query optimiser
- allowing the DB to be run on separate hardware

For information on how to install and use PostgreSQL in Synapse, please see [Using Postgres](#)

SQLite is only acceptable for testing purposes. SQLite should not be used in a production server. Synapse will perform poorly when using SQLite, especially when participating in large rooms.

TLS certificates

The default configuration exposes a single HTTP port on the local interface:

`http://localhost:8008`. It is suitable for local testing, but for any practical use, you will need Synapse's APIs to be served over HTTPS.

The recommended way to do so is to set up a reverse proxy on port `8448`. You can find documentation on doing so in [the reverse proxy documentation](#).

Alternatively, you can configure Synapse to expose an HTTPS port. To do so, you will need to edit `homeserver.yaml`, as follows:

- First, under the `listeners` option, add the configuration for the TLS-enabled listener like so:

```
listeners:  
  - port: 8448  
    type: http  
    tls: true  
    resources:  
      - names: [client, federation]
```

- You will also need to add the options `tls_certificate_path` and `tls_private_key_path` to your configuration file. You will need to manage provisioning of these certificates yourself.
- You can find more information about these options as well as how to configure synapse in the [configuration manual](#).

If you are using your own certificate, be sure to use a `.pem` file that includes the full certificate chain including any intermediate certificates (for instance, if using certbot, use `fullchain.pem` as your certificate, not `cert.pem`).

For a more detailed guide to configuring your server for federation, see [Federation](#).

Client Well-Known URI

Setting up the client Well-Known URI is optional but if you set it up, it will allow users to enter their full username (e.g. `@user:<server_name>`) into clients which support well-known

lookup to automatically configure the homeserver and identity server URLs. This is useful so that users don't have to memorize or think about the actual homeserver URL you are using.

The URL `https://<server_name>/.well-known/matrix/client` should return JSON in the following format.

```
{
  "m.homeserver": {
    "base_url": "https://<matrix.example.com>"
  }
}
```

It can optionally contain identity server information as well.

```
{
  "m.homeserver": {
    "base_url": "https://<matrix.example.com>"
  },
  "m.identity_server": {
    "base_url": "https://<identity.example.com>"
  }
}
```

To work in browser based clients, the file must be served with the appropriate Cross-Origin Resource Sharing (CORS) headers. A recommended value would be `Access-Control-Allow-Origin: *` which would allow all browser based clients to view it.

In nginx this would be something like:

```
location /.well-known/matrix/client {
  return 200 '{
    "m.homeserver": {
      "base_url": "https://<matrix.example.com>"
    }
  }';
  default_type application/json;
  add_header Access-Control-Allow-Origin *;
}
```

You should also ensure the `public_baseurl` option in `homeserver.yaml` is set correctly. `public_baseurl` should be set to the URL that clients will use to connect to your server. This is the same URL you put for the `m.homeserver` `base_url` above.

```
public_baseurl: "https://<matrix.example.com>"
```

Email

It is desirable for Synapse to have the capability to send email. This allows Synapse to send password reset emails, send verifications when an email address is added to a user's account, and send email notifications to users when they receive new messages.

To configure an SMTP server for Synapse, modify the configuration section headed `email`, and be sure to have at least the `smtp_host`, `smtp_port` and `notif_from` fields filled out. You may also need to set `smtp_user`, `smtp_pass`, and `require_transport_security`.

If email is not configured, password reset, registration and notifications via email will be disabled.

Registering a user

One way to create a new user is to do so from a client like [Element](#). This requires registration to be enabled via the `enable_registration` setting.

Alternatively, you can create new users from the command line. This can be done as follows:

1. If synapse was installed via pip, activate the virtualenv as follows (if Synapse was installed via a prebuilt package, `register_new_matrix_user` should already be on the search path):

```
cd ~/synapse
source env/bin/activate
synctl start # if not already running
```

2. Run the following command:

```
register_new_matrix_user -c homeserver.yaml
```

This will prompt you to add details for the new user, and will then connect to the running Synapse to create the new user. For example:

```
New user localpart: erikj
Password:
Confirm password:
Make admin [no]:
Success!
```

This process uses a setting `registration_shared_secret`, which is shared between Synapse itself and the `register_new_matrix_user` script. It doesn't matter what it is (a random value is generated by `--generate-config`), but it should be kept secret, as anyone with knowledge of it can register users, including admin accounts, on your server even if `enable_registration` is `false`.

Setting up a TURN server

For reliable VoIP calls to be routed via this homeserver, you **MUST** configure a TURN server. See [TURN setup](#) for details.

URL previews

Synapse includes support for previewing URLs, which is disabled by default. To turn it on you must enable the `url_preview_enabled: True` config parameter and explicitly specify the IP ranges that Synapse is not allowed to spider for previewing in the `url_preview_ip_range_blacklist` configuration parameter. This is critical from a security perspective to stop arbitrary Matrix users spidering 'internal' URLs on your network. At the very least we recommend that your loopback and RFC1918 IP addresses are blacklisted.

This also requires the optional `lxml` python dependency to be installed. This in turn requires the `libxml2` library to be available - on Debian/Ubuntu this means `apt-get install libxml2-dev`, or equivalent for your OS.

Backups

Don't forget to take [backups](#) of your new server!

Troubleshooting Installation

`pip` seems to leak *lots* of memory during installation. For instance, a Linux host with 512MB of RAM may run out of memory whilst installing Twisted. If this happens, you will have to individually install the dependencies which are failing, e.g.:

```
pip install twisted
```

If you have any other problems, feel free to ask in [#synapse:matrix.org](#).

Using Postgres

The minimum supported version of PostgreSQL is determined by the [Dependency Deprecation Policy](#).

Install postgres client libraries

Synapse will require the python postgres client library in order to connect to a postgres database.

- If you are using the [matrix.org debian/ubuntu packages](#), the necessary python library will already be installed, but you will need to ensure the low-level postgres library is installed, which you can do with `apt install libpq5`.
- For other pre-built packages, please consult the documentation from the relevant package.
- If you installed synapse [in a virtualenv](#), you can install the library with:

```
~/synapse/env/bin/pip install "matrix-synapse[postgres]"
```

(substituting the path to your virtualenv for `~/synapse/env`, if you used a different path). You will require the postgres development files. These are in the `libpq-dev` package on Debian-derived distributions.

Set up database

Assuming your PostgreSQL database user is called `postgres`, first authenticate as the database user with:

```
su - postgres
# Or, if your system uses sudo to get administrative rights
sudo -u postgres bash
```

Then, create a postgres user and a database with:

```
# this will prompt for a password for the new user
createuser --pwprompt synapse_user

createdb --encoding=UTF8 --locale=C --template=template0 --owner=synapse_user
synapse
```

The above will create a user called `synapse_user`, and a database called `synapse`.

Note that the PostgreSQL database *must* have the correct encoding set (as shown above), otherwise it will not be able to store UTF8 strings.

You may need to enable password authentication so `synapse_user` can connect to the database. See <https://www.postgresql.org/docs/current/auth-pg-hba-conf.html>.

Synapse config

When you are ready to start using PostgreSQL, edit the `database` section in your config file to match the following lines:

```
database:
  name: psycopg2
  args:
    user: <user>
    password: <pass>
    dbname: <db>
    host: <host>
    cp_min: 5
    cp_max: 10
```

All key, values in `args` are passed to the `psycopg2.connect(..)` function, except keys beginning with `cp_`, which are consumed by the twisted adbapi connection pool. See the [libpq documentation](#) for a list of options which can be passed.

You should consider tuning the `args.keepalives_*` options if there is any danger of the connection between your homeserver and database dropping, otherwise Synapse may block for an extended period while it waits for a response from the database server. Example values might be:

```
database:
  args:
    # ... as above

    # seconds of inactivity after which TCP should send a keepalive message to
    # the server
    keepalives_idle: 10

    # the number of seconds after which a TCP keepalive message that is not
    # acknowledged by the server should be retransmitted
    keepalives_interval: 10

    # the number of TCP keepalives that can be lost before the client's
    # connection
    # to the server is considered dead
    keepalives_count: 3
```

Backups

Don't forget to [back up](#) your database!

Tuning Postgres

The default settings should be fine for most deployments. For larger scale deployments tuning some of the settings is recommended, details of which can be found at https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server.

In particular, we've found tuning the following values helpful for performance:

- `shared_buffers`
- `effective_cache_size`
- `work_mem`
- `maintenance_work_mem`
- `autovacuum_work_mem`

Note that the appropriate values for those fields depend on the amount of free memory the database host has available.

Additionally, admins of large deployments might want to consider using huge pages to help manage memory, especially when using large values of `shared_buffers`. You can read more about that [here](#).

Porting from SQLite

Overview

The script `synapse_port_db` allows porting an existing synapse server backed by SQLite to using PostgreSQL. This is done as a two phase process:

1. Copy the existing SQLite database to a separate location and run the port script against that offline database.
2. Shut down the server. Rerun the port script to port any data that has come in since taking the first snapshot. Restart server against the PostgreSQL database.

The port script is designed to be run repeatedly against newer snapshots of the SQLite database file. This makes it safe to repeat step 1 if there was a delay between taking the previous snapshot and being ready to do step 2.

It is safe to at any time kill the port script and restart it.

However, under no circumstances should the SQLite database be `VACUUM`ed between multiple runs of the script. Doing so can lead to an inconsistent copy of your database into Postgres. To avoid accidental error, the script will check that SQLite's `auto_vacuum` mechanism is disabled, but the script is not able to protect against a manual `VACUUM` operation performed either by the administrator or by any automated task that the administrator may have configured.

Note that the database may take up significantly more (25% - 100% more) space on disk after porting to Postgres.

Using the port script

Firstly, shut down the currently running synapse server and copy its database file (typically `homeserver.db`) to another location. Once the copy is complete, restart synapse. For instance:

```
synctl stop
cp homeserver.db homeserver.db.snapshot
synctl start
```

Copy the old config file into a new config file:

```
cp homeserver.yaml homeserver-postgres.yaml
```

Edit the database section as described in the section *Synapse config* above and with the SQLite snapshot located at `homeserver.db.snapshot` simply run:

```
synapse_port_db --sqlite-database homeserver.db.snapshot \
--postgres-config homeserver-postgres.yaml
```

The flag `--curses` displays a coloured curses progress UI. (NOTE: if your terminal is too small the script will error out)

If the script took a long time to complete, or time has otherwise passed since the original snapshot was taken, repeat the previous steps with a newer snapshot.

To complete the conversion shut down the synapse server and run the port script one last time, e.g. if the SQLite database is at `homeserver.db` run:

```
synapse_port_db --sqlite-database homeserver.db \
--postgres-config homeserver-postgres.yaml
```

Once that has completed, change the synapse config to point at the PostgreSQL database configuration file `homeserver-postgres.yaml`:

```
synctl stop
mv homeserver.yaml homeserver-old-sqlite.yaml
mv homeserver-postgres.yaml homeserver.yaml
synctl start
```

Synapse should now be running against PostgreSQL.

Troubleshooting

Alternative auth methods

If you get an error along the lines of `FATAL: Ident authentication failed for user "synapse_user"`, you may need to use an authentication method other than `ident`:

- If the `synapse_user` user has a password, add the password to the `database:` section of `homeserver.yaml`. Then add the following to `pg_hba.conf`:

```
host      synapse      synapse_user      ::1/128      md5  # or `scram-sha-256`  
instead of `md5` if you use that
```

- If the `synapse_user` user does not have a password, then a password doesn't have to be added to `homeserver.yaml`. But the following does need to be added to `pg_hba.conf`:

```
host      synapse      synapse_user      ::1/128      trust
```

Note that line order matters in `pg_hba.conf`, so make sure that if you do add a new line, it is inserted before:

```
host      all          all          ::1/128      ident
```

Fixing incorrect `COLLATE` or `CTYPE`

Synapse will refuse to start when using a database with incorrect values of `COLLATE` and `CTYPE` unless the config flag `allow_unsafe_locale`, found in the `database` section of the config, is set to true. Using different locales can cause issues if the locale library is updated from underneath the database, or if a different version of the locale is used on any replicas.

If you have a database with an unsafe locale, the safest way to fix the issue is to dump the database and recreate it with the correct locale parameter (as shown above). It is also possible to change the parameters on a live database and run a `REINDEX` on the entire database, however extreme care must be taken to avoid database corruption.

Note that the above may fail with an error about duplicate rows if corruption has already occurred, and such duplicate rows will need to be manually removed.

Using a reverse proxy with Synapse

It is recommended to put a reverse proxy such as [nginx](#), [Apache](#), [Caddy](#), [HAProxy](#) or [relayd](#) in front of Synapse. One advantage of doing so is that it means that you can expose the default https port (443) to Matrix clients without needing to run Synapse with root privileges.

You should configure your reverse proxy to forward requests to `/_matrix` or `/_synapse/client` to Synapse, and have it set the `X-Forwarded-For` and `X-Forwarded-Proto` request headers.

You should remember that Matrix clients and other Matrix servers do not necessarily need to connect to your server via the same server name or port. Indeed, clients will use port 443 by default, whereas servers default to port 8448. Where these are different, we refer to the 'client port' and the 'federation port'. See [the Matrix specification](#) for more details of the algorithm used for federation connections, and [Delegation](#) for instructions on setting up delegation.

NOTE: Your reverse proxy must not `canonicalise` or `normalise` the requested URI in any way (for example, by decoding `%xx` escapes). Beware that Apache *will* canonicalise URIs unless you specify `nocanon`.

Let's assume that we expect clients to connect to our server at `https://matrix.example.com`, and other servers to connect at `https://example.com:8448`. The following sections detail the configuration of the reverse proxy and the homeserver.

Homeserver Configuration

The HTTP configuration will need to be updated for Synapse to correctly record client IP addresses and generate redirect URLs while behind a reverse proxy.

In `homeserver.yaml` set `x_forwarded: true` in the port 8008 section and consider setting `bind_addresses: ['127.0.0.1']` so that the server only listens to traffic on localhost. (Do not change `bind_addresses` to `127.0.0.1` when using a containerized Synapse, as that will prevent it from responding to proxied traffic.)

Optionally, you can also set `request_id_header` so that the server extracts and re-uses the same request ID format that the reverse proxy is using.

Reverse-proxy configuration examples

NOTE: You only need one of these.

nginx

```

server {
    listen 443 ssl;
    listen [::]:443 ssl;

    # For the federation port
    listen 8448 ssl default_server;
    listen [::]:8448 ssl default_server;

    server_name matrix.example.com;

    location ~ ^(/_matrix|/_synapse/client) {
        # note: do not add a path (even a single /) after the port in
        `proxy_pass`,
        # otherwise nginx will canonicalise the URI and cause signature
        verification
        # errors.
        proxy_pass http://localhost:8008;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Host $host:$server_port;

        # Nginx by default only allows file uploads up to 1M in size
        # Increase client_max_body_size to match max_upload_size defined in
        homeserver.yaml
        client_max_body_size 50M;

        # Synapse responses may be chunked, which is an HTTP/1.1 feature.
        proxy_http_version 1.1;
    }
}

```

Caddy v2

```

matrix.example.com {
    reverse_proxy /_matrix/* localhost:8008
    reverse_proxy /_synapse/client/* localhost:8008
}

example.com:8448 {
    reverse_proxy /_matrix/* localhost:8008
}

```

Delegation example:

```

example.com {
  header /.well-known/matrix/* Content-Type application/json
  header /.well-known/matrix/* Access-Control-Allow-Origin *
  respond /.well-known/matrix/server `>{"m.server": "matrix.example.com:443"}` 
  respond /.well-known/matrix/client `{"m.homeserver": 
  {"base_url": "https://matrix.example.com"}, "m.identity_server": 
  {"base_url": "https://identity.example.com"}}` 
}

matrix.example.com {
  reverse_proxy /_matrix/* localhost:8008
  reverse_proxy /_synapse/client/* localhost:8008
}

```

Apache

```

<VirtualHost *:443>
  SSLEngine on
  ServerName matrix.example.com

  RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
  AllowEncodedSlashes NoDecode
  ProxyPreserveHost on
  ProxyPass /_matrix http://127.0.0.1:8008/_matrix nocanon
  ProxyPassReverse /_matrix http://127.0.0.1:8008/_matrix
  ProxyPass /_synapse/client http://127.0.0.1:8008/_synapse/client nocanon
  ProxyPassReverse /_synapse/client http://127.0.0.1:8008/_synapse/client
</VirtualHost>

<VirtualHost *:8448>
  SSLEngine on
  ServerName example.com

  RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
  AllowEncodedSlashes NoDecode
  ProxyPass /_matrix http://127.0.0.1:8008/_matrix nocanon
  ProxyPassReverse /_matrix http://127.0.0.1:8008/_matrix
</VirtualHost>

```

NOTE: ensure the `nocanon` options are included.

NOTE 2: It appears that Synapse is currently incompatible with the ModSecurity module for Apache (`mod_security2`). If you need it enabled for other services on your web server, you can disable it for Synapse's two VirtualHosts by including the following lines before each of the two `</VirtualHost>` above:

```

<IfModule security2_module>
  SecRuleEngine off
</IfModule>

```

NOTE 3: Missing `ProxyPreserveHost on` can lead to a redirect loop.

HAProxy

```

frontend https
  bind *:443,[:]:443 ssl crt /etc/ssl/haproxy/ strict-sni alpn h2,http/1.1
  http-request set-header X-Forwarded-Proto https if { ssl_fc }
  http-request set-header X-Forwarded-Proto http if !{ ssl_fc }
  http-request set-header X-Forwarded-For %[src]

  # Matrix client traffic
  acl matrix-host hdr(host) -i matrix.example.com matrix.example.com:443
  acl matrix-path path_beg /_matrix
  acl matrix-path path_beg /_synapse/client

  use_backend matrix if matrix-host matrix-path

frontend matrix-federation
  bind *:8448,[:]:8448 ssl crt /etc/ssl/haproxy/synapse.pem alpn h2,http/1.1
  http-request set-header X-Forwarded-Proto https if { ssl_fc }
  http-request set-header X-Forwarded-Proto http if !{ ssl_fc }
  http-request set-header X-Forwarded-For %[src]

  default_backend matrix

backend matrix
  server matrix 127.0.0.1:8008

```

Example configuration, if using a UNIX socket. The configuration lines regarding the frontends do not need to be modified.

```

backend matrix
  server matrix unix@/run/synapse/main_public.sock

```

Example configuration when using a single port for both client and federation traffic.

```

frontend https
  bind *:443,[:]:443 ssl crt /etc/ssl/haproxy/ strict-sni alpn h2,http/1.1
  http-request set-header X-Forwarded-Proto https if { ssl_fc }
  http-request set-header X-Forwarded-Proto http if !{ ssl_fc }
  http-request set-header X-Forwarded-For %[src]

  acl matrix-host hdr(host) -i matrix.example.com matrix.example.com:443
  acl matrix-sni ssl_fc_sni matrix.example.com
  acl matrix-path path_beg /_matrix
  acl matrix-path path_beg /_synapse/client

  use_backend matrix if matrix-host matrix-path
  use_backend matrix if matrix-sni

backend matrix
  server matrix 127.0.0.1:8008

```

[Delegation](#) example:

```
frontend https
  acl matrix-well-known-client-path path /.well-known/matrix/client
  acl matrix-well-known-server-path path /.well-known/matrix/server
  use_backend matrix-well-known-client if matrix-well-known-client-path
  use_backend matrix-well-known-server if matrix-well-known-server-path

backend matrix-well-known-client
  http-after-response set-header Access-Control-Allow-Origin "*"
  http-after-response set-header Access-Control-Allow-Methods "GET, POST, PUT, DELETE, OPTIONS"
  http-after-response set-header Access-Control-Allow-Headers "Origin, X-Requested-With, Content-Type, Accept, Authorization"
  http-request return status 200 content-type application/json string
  '{"m.homeserver": {"base_url": "https://matrix.example.com"}, "m.identity_server": {"base_url": "https://identity.example.com"}}'

backend matrix-well-known-server
  http-after-response set-header Access-Control-Allow-Origin "*"
  http-after-response set-header Access-Control-Allow-Methods "GET, POST, PUT, DELETE, OPTIONS"
  http-after-response set-header Access-Control-Allow-Headers "Origin, X-Requested-With, Content-Type, Accept, Authorization"
  http-request return status 200 content-type application/json string
  '{"m.server": "matrix.example.com:443"}'
```

Relayd

```

table <webserver>    { 127.0.0.1 }
table <matrixserver> { 127.0.0.1 }

http protocol "https" {
    tls { no tlsv1.0, ciphers "HIGH" }
    tls keypair "example.com"
    match header set "X-Forwarded-For"    value "$REMOTE_ADDR"
    match header set "X-Forwarded-Proto" value "https"

    # set CORS header for .well-known/matrix/server, .well-known/matrix/client
    # httpd does not support setting headers, so do it here
    match request path "/.well-known/matrix/*" tag "matrix-cors"
    match response tagged "matrix-cors" header set "Access-Control-Allow-Origin"
    value "*"

    pass quick path "/_matrix/*"           forward to <matrixserver>
    pass quick path "/_synapse/client/*"  forward to <matrixserver>

    # pass on non-matrix traffic to webserver
    pass                                forward to <webserver>
}

relay "https_traffic" {
    listen on egress port 443 tls
    protocol "https"
    forward to <matrixserver> port 8008 check tcp
    forward to <webserver>    port 8080 check tcp
}

http protocol "matrix" {
    tls { no tlsv1.0, ciphers "HIGH" }
    tls keypair "example.com"
    block
    pass quick path "/_matrix/*"           forward to <matrixserver>
    pass quick path "/_synapse/client/*"  forward to <matrixserver>
}

relay "matrix_federation" {
    listen on egress port 8448 tls
    protocol "matrix"
    forward to <matrixserver> port 8008 check tcp
}

```

Health check endpoint

Synapse exposes a health check endpoint for use by reverse proxies. Each configured HTTP listener has a `/health` endpoint which always returns 200 OK (and doesn't get logged).

Synapse administration endpoints

Endpoints for administering your Synapse instance are placed under `/_synapse/admin`. These require authentication through an access token of an admin user. However as access to these endpoints grants the caller a lot of power, we do not recommend exposing them to the public internet without good reason.

Using a forward proxy with Synapse

You can use Synapse with a forward or outbound proxy. An example of when this is necessary is in corporate environments behind a DMZ (demilitarized zone). Synapse supports routing outbound HTTP(S) requests via a proxy. Only HTTP(S) proxy is supported, not SOCKS proxy or anything else.

Configure

The `http_proxy`, `https_proxy`, `no_proxy` environment variables are used to specify proxy settings. The environment variable is not case sensitive.

- `http_proxy`: Proxy server to use for HTTP requests.
- `https_proxy`: Proxy server to use for HTTPS requests.
- `no_proxy`: Comma-separated list of hosts, IP addresses, or IP ranges in CIDR format which should not use the proxy. Synapse will directly connect to these hosts.

The `http_proxy` and `https_proxy` environment variables have the form: `[scheme://] [<username>:<password>@] <host>[:<port>]`

- Supported schemes are `http://` and `https://`. The default scheme is `http://` for compatibility reasons; it is recommended to set a scheme. If scheme is set to `https://` the connection uses TLS between Synapse and the proxy.

NOTE: Synapse validates the certificates. If the certificate is not valid, then the connection is dropped.

- Default port if not given is `1080`.
- Username and password are optional and will be used to authenticate against the proxy.

Examples

- `HTTP_PROXY=http://USERNAME:PASSWORD@10.0.1.1:8080/`
- `HTTPS_PROXY=http://USERNAME:PASSWORD@proxy.example.com:8080/`
- `NO_PROXY=master.hostname.example.com,10.1.0.0/16,172.30.0.0/16`

NOTE: Synapse does not apply the IP blacklist to connections through the proxy (since the DNS resolution is done by the proxy). It is expected that the proxy or firewall will apply blacklisting of IP addresses.

Connection types

The proxy will be **used** for:

- push
- url previews
- phone-home stats
- re captcha validation
- CAS auth validation
- OpenID Connect
- Outbound federation
- Federation (checking public key revocation)
- Fetching public keys of other servers
- Downloading remote media

It will **not be used** for:

- Application Services
- Identity servers
- In worker configurations
 - connections between workers
 - connections from workers to Redis

Troubleshooting

If a proxy server is used with TLS (HTTPS) and no connections are established, it is most likely due to the proxy's certificates. To test this, the validation in Synapse can be deactivated.

NOTE: This has an impact on security and is for testing purposes only!

To deactivate the certificate validation, the following setting must be added to your `homeserver.yaml`.

```
use_insecure_ssl_client_just_for_testing_do_not_use: true
```

Overview

This document explains how to enable VoIP relaying on your homeserver with TURN.

The synapse Matrix homeserver supports integration with TURN server via the [TURN server REST API](#). This allows the homeserver to generate credentials that are valid for use on the TURN server through the use of a secret shared between the homeserver and the TURN server.

This documentation provides two TURN server configuration examples:

- [coturn](#)
- [eturnal](#)

Requirements

For TURN relaying to work, the TURN service must be hosted on a server/endpoint with a public IP.

Hosting TURN behind NAT requires port forwarding and for the NAT gateway to have a public IP. However, even with appropriate configuration, NAT is known to cause issues and to often not work.

Afterwards, the homeserver needs some further configuration.

Synapse setup

Your homeserver configuration file needs the following extra keys:

1. `turn_uris`
2. `turn_shared_secret`
3. `turn_user_lifetime`
4. `turn_allow_guests`

As an example, here is the relevant section of the config file for `matrix.org`. The `turn_uris` are appropriate for TURN servers listening on the default ports, with no TLS.

```
turn_uris: [ "turn:turn.matrix.org?transport=udp", "turn:turn.matrix.org?transport=tcp" ]
turn_shared_secret: "n0t4ctuAllymatr1Xd0TorgSshar3d5ecret4obvIousreAsons"
turn_user_lifetime: 86400000
turn_allow_guests: true
```

After updating the homeserver configuration, you must restart synapse:

- If you use synctl:

```
# Depending on how Synapse is installed, synctl may already be on
# your PATH. If not, you may need to activate a virtual environment.
synctl restart
```

- If you use systemd:

```
systemctl restart matrix-synapse.service
```

... and then reload any clients (or wait an hour for them to refresh their settings).

Troubleshooting

The normal symptoms of a misconfigured TURN server are that calls between devices on different networks ring, but get stuck at "call connecting". Unfortunately, troubleshooting this can be tricky.

Here are a few things to try:

- Check that you have opened your firewall to allow TCP and UDP traffic to the TURN ports (normally 3478 and 5349).
- Check that you have opened your firewall to allow UDP traffic to the UDP relay ports (49152-65535 by default).
- Try disabling TLS/DTLS listeners and enable only its (unencrypted) TCP/UDP listeners. (This will only leave signaling traffic unencrypted; voice & video WebRTC traffic is always encrypted.)
- Some WebRTC implementations (notably, that of Google Chrome) appear to get confused by TURN servers which are reachable over IPv6 (this appears to be an unexpected side-effect of its handling of multiple IP addresses as defined by [draft-ietf-rtcweb-ip-handling](#)).

Try removing any AAAA records for your TURN server, so that it is only reachable over IPv4.

- If your TURN server is behind NAT:

- double-check that your NAT gateway is correctly forwarding all TURN ports (normally 3478 & 5349 for TCP & UDP TURN traffic, and 49152-65535 for the UDP relay) to the NAT-internal address of your TURN server. If advertising both IPv4 and IPv6 external addresses via the `external-ip` option, ensure that the NAT is

forwarding both IPv4 and IPv6 traffic to the IPv4 and IPv6 internal addresses of your TURN server. When in doubt, remove AAAA records for your TURN server and specify only an IPv4 address as your `external-ip`.

- ensure that your TURN server uses the NAT gateway as its default route.
- Enable more verbose logging, in `coturn` via the `verbose` setting:

```
verbose
```

or with `eternal` with the shell command `eternalctl loglevel debug` or in the configuration file (the service needs to `reload` for it to become effective):

```
## Logging configuration:
log_level: debug
```

... and then see if there are any clues in its logs.

- If you are using a browser-based client under Chrome, check `chrome://webrtc-internals/` for insights into the internals of the negotiation. On Firefox, check the "Connection Log" on `about:webrtc`.

(Understanding the output is beyond the scope of this document!)

- You can test your Matrix homeserver TURN setup with <https://test.voip.librepush.net/>. Note that this test is not fully reliable yet, so don't be discouraged if the test fails. [Here](#) is the github repo of the source of the tester, where you can file bug reports.
- There is a WebRTC test tool at <https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/>. To use it, you will need a username/password for your TURN server. You can either:
 - look for the `GET /_matrix/client/r0/voip/turnServer` request made by a matrix client to your homeserver in your browser's network inspector. In the response you should see `username` and `password`. Or:
 - Use the following shell commands for `coturn`:

```
secret=staticAuthSecretHere

u=$((`date +%s` + 3600)):test
p=$(echo -n $u | openssl dgst -hmac $secret -sha1 -binary | base64)
echo -e "username: $u\npassword: $p"
```

or for `eternal`

```
eternalctl credentials
```

- Or (**coturn only**): Temporarily configure `coturn` to accept a static username/password. To do this, comment out `use-auth-secret` and `static-auth-secret` and add the following:

```
lt-cred-mech
user=username:password
```

Note: these settings will not take effect unless `use-auth-secret` and `static-auth-secret` are disabled.

Restart coturn after changing the configuration file.

Remember to restore the original settings to go back to testing with Matrix clients!

If the TURN server is working correctly, you should see at least one `relay` entry in the results.

coturn TURN server

The following sections describe how to install **coturn** (which implements the TURN REST API).

coturn setup

Initial installation

The TURN daemon **coturn** is available from a variety of sources such as native package managers, or installation from source.

Debian and Ubuntu based distributions

Just install the debian package:

```
sudo apt install coturn
```

This will install and start a systemd service called **coturn**.

Source installation

1. Download the [latest release](#) from github. Unpack it and `cd` into the directory.
2. Configure it:

```
./configure
```

You may need to install `libevent2`: if so, you should do so in the way recommended by your operating system. You can ignore warnings about lack of database support: a database is unnecessary for this purpose.

3. Build and install it:

```
make
sudo make install
```

Configuration

1. Create or edit the config file in `/etc/turnserver.conf`. The relevant lines, with example values, are:

```
use-auth-secret
static-auth-secret=[your secret key here]
realm=turn.myserver.org
```

See `turnserver.conf` for explanations of the options. One way to generate the `static-auth-secret` is with `pwgen`:

```
pwgen -s 64 1
```

A `realm` must be specified, but its value is somewhat arbitrary. (It is sent to clients as part of the authentication flow.) It is conventional to set it to be your server name.

2. You will most likely want to configure `coturn` to write logs somewhere. The easiest way is normally to send them to the syslog:

```
syslog
```

(in which case, the logs will be available via `journalctl -u coturn` on a systemd system). Alternatively, `coturn` can be configured to write to a logfile - check the example config file supplied with `coturn`.

3. Consider your security settings. TURN lets users request a relay which will connect to arbitrary IP addresses and ports. The following configuration is suggested as a minimum starting point:

```

# VoIP traffic is all UDP. There is no reason to let users connect to
arbitrary TCP endpoints via the relay.
no-tcp-relay

# don't let the relay ever try to connect to private IP address ranges
within your network (if any)
# given the turn server is likely behind your firewall, remember to include
any privileged public IPs too.
denied-peer-ip=10.0.0.0-10.255.255.255
denied-peer-ip=192.168.0.0-192.168.255.255
denied-peer-ip=172.16.0.0-172.31.255.255

# recommended additional local peers to block, to mitigate external access
to internal services.
# https://www rtcsec com/article/slack-webrtc-turn-compromise-and-bug-bounty/#how-to-fix-an-open-turn-relay-to-address-this-vulnerability
no-multicast-peers
denied-peer-ip=0.0.0.0-0.255.255.255
denied-peer-ip=100.64.0.0-100.127.255.255
denied-peer-ip=127.0.0.0-127.255.255.255
denied-peer-ip=169.254.0.0-169.254.255.255
denied-peer-ip=192.0.0.0-192.0.0.255
denied-peer-ip=192.0.2.0-192.0.2.255
denied-peer-ip=192.88.99.0-192.88.99.255
denied-peer-ip=198.18.0.0-198.19.255.255
denied-peer-ip=198.51.100.0-198.51.100.255
denied-peer-ip=203.0.113.0-203.0.113.255
denied-peer-ip=240.0.0.0-255.255.255.255

# special case the turn server itself so that client->TURN->TURN->client
flows work
# this should be one of the turn server's listening IPs
allowed-peer-ip=10.0.0.1

# consider whether you want to limit the quota of relayed streams per user
(or total) to avoid risk of DoS.
user-quota=12 # 4 streams per video call, so 12 streams = 3 simultaneous
relayed calls per user.
total-quota=1200

```

4. Also consider supporting TLS/DTLS. To do this, add the following settings to `turnserver.conf`:

```
# TLS certificates, including intermediate certs.
# For Let's Encrypt certificates, use `fullchain.pem` here.
cert=/path/to/fullchain.pem

# TLS private key file
pkey=/path/to/privkey.pem

# Ensure the configuration lines that disable TLS/DTLS are commented-out or
# removed
#no-tls
#no-dtls
```

In this case, replace the `turn:` schemes in the `turn_uris` settings below with `turns:`.

We recommend that you only try to set up TLS/DTLS once you have set up a basic installation and got it working.

NB: If your TLS certificate was provided by Let's Encrypt, TLS/DTLS will not work with any Matrix client that uses Chromium's WebRTC library. This currently includes Element Android & iOS; for more details, see their [respective issues](#) as well as the underlying [WebRTC issue](#). Consider using a ZeroSSL certificate for your TURN server as a working alternative.

5. Ensure your firewall allows traffic into the TURN server on the ports you've configured it to listen on (By default: 3478 and 5349 for TURN traffic (remember to allow both TCP and UDP traffic), and ports 49152-65535 for the UDP relay.)
6. If your TURN server is behind NAT, the NAT gateway must have an external, publicly-reachable IP address. You must configure `coturn` to advertise that address to connecting clients:

```
external-ip=EXTERNAL_NAT_IPv4_ADDRESS
```

You may optionally limit the TURN server to listen only on the local address that is mapped by NAT to the external address:

```
listening-ip=INTERNAL_TURNSERVER_IPv4_ADDRESS
```

If your NAT gateway is reachable over both IPv4 and IPv6, you may configure `coturn` to advertise each available address:

```
external-ip=EXTERNAL_NAT_IPv4_ADDRESS
external-ip=EXTERNAL_NAT_IPv6_ADDRESS
```

When advertising an external IPv6 address, ensure that the firewall and network settings of the system running your TURN server are configured to accept IPv6 traffic,

and that the TURN server is listening on the local IPv6 address that is mapped by NAT to the external IPv6 address.

7. (Re)start the turn server:

- o If you used the Debian package (or have set up a systemd unit yourself):

```
sudo systemctl restart coturn
```

- o If you built from source:

```
/usr/local/bin/turnserver -o
```

eturnal TURN server

The following sections describe how to install `eturnal` (which implements the TURN REST API).

eturnal setup

Initial installation

The `eturnal` TURN server implementation is available from a variety of sources such as native package managers, binary packages, installation from source or [container image](#). They are all described [here](#).

Quick-Test instructions in a [Linux Shell](#) or with [Docker](#) are available as well.

Configuration

After installation, `eturnal` usually ships a [default configuration file](#) here: `/etc/eturnal.yml` (and, if not found there, there is a backup file here: `/opt/eturnal/etc/eturnal.yml`). It uses the (indentation-sensitive!) [YAML](#) format. The file contains further explanations.

Here are some hints how to configure `eturnal` on your [host machine](#) or when using e.g. [Docker](#). You may also further deep dive into the [reference documentation](#).

`eturnal` runs out of the box with the default configuration. To enable TURN and to integrate it with your homeserver, some aspects in `eturnal`'s default configuration file must be edited:

1. Homeserver's `turn_shared_secret` and `eturnal`'s shared `secret` for authentication

Both need to have the same value. Uncomment and adjust this line in `eturnal`'s configuration file:

```
secret: "long-and-cryptic"      # Shared secret, CHANGE THIS.
```

One way to generate a `secret` is with `pwgen`:

```
pwgen -s 64 1
```

2. Public IP address

If your TURN server is behind NAT, the NAT gateway must have an external, publicly-reachable IP address. `eternal` tries to autodetect the public IP address, however, it may also be configured by uncommenting and adjusting this line, so `eternal` advertises that address to connecting clients:

```
relay_ipv4_addr: "203.0.113.4" # The server's public IPv4 address.
```

If your NAT gateway is reachable over both IPv4 and IPv6, you may configure `eternal` to advertise each available address:

```
relay_ipv4_addr: "203.0.113.4" # The server's public IPv4 address.
relay_ipv6_addr: "2001:db8::4" # The server's public IPv6 address
(optional).
```

When advertising an external IPv6 address, ensure that the firewall and network settings of the system running your TURN server are configured to accept IPv6 traffic, and that the TURN server is listening on the local IPv6 address that is mapped by NAT to the external IPv6 address.

3. Logging

If `eternal` was started by systemd, log files are written into the `/var/log/eternal` directory by default. In order to log to the `journal` instead, the `log_dir` option can be set to `stdout` in the configuration file.

4. Security considerations

Consider your security settings. TURN lets users request a relay which will connect to arbitrary IP addresses and ports. The following configuration is suggested as a minimum starting point, [see also the official documentation](#):

```
## Reject TURN relaying from/to the following addresses/networks:
blacklist:                      # This is the default blacklist.
  - "127.0.0.0/8"                # IPv4 loopback.
  - "::1"                        # IPv6 loopback.
  - recommended                  # Expands to a number of networks recommended to
be
                                         # blocked, but includes private networks. Those
                                         # would have to be 'whitelist'ed if eternal
serves
                                         # local clients/peers within such networks.
```

To whitelist IP addresses or specific (private) networks, you need to **add** a whitelist part into the configuration file, e.g.:

```
whitelist:
  - "192.168.0.0/16"
  - "203.0.113.113"
  - "2001:db8::/64"
```

The more specific, the better.

5. TURNS (TURN via TLS/DTLS)

Also consider supporting TLS/DTLS. To do this, adjust the following settings in the `eternal.yml` configuration file (TLS parts should not be commented anymore):

```
listen:
  - ip: "::"
    port: 3478
    transport: udp
  - ip: "::"
    port: 3478
    transport: tcp
  - ip: "::"
    port: 5349
    transport: tls

## TLS certificate/key files (must be readable by 'eternal' user!):
tls_crt_file: /etc/eternal/tls/crt.pem
tls_key_file: /etc/eternal/tls/key.pem
```

In this case, replace the `turn:` schemes in homeserver's `turn_uris` settings with `turns:`. More is described [here](#).

We recommend that you only try to set up TLS/DTLS once you have set up a basic installation and got it working.

NB: If your TLS certificate was provided by Let's Encrypt, TLS/DTLS will not work with any Matrix client that uses Chromium's WebRTC library. This currently includes Element Android & iOS; for more details, see their [respective issues](#) as well as the underlying [WebRTC issue](#). Consider using a ZeroSSL certificate for your TURN server as a working alternative.

6. Firewall

Ensure your firewall allows traffic into the TURN server on the ports you've configured it to listen on (By default: 3478 and 5349 for TURN traffic (remember to allow both TCP and UDP traffic), and ports 49152-65535 for the UDP relay.)

7. Reload/ restarting `eternal`

Changes in the configuration file require `eternal` to reload/ restart, this can be achieved by:

```
eternalctl reload
```

`eternal` performs a configuration check before actually reloading/ restarting and provides hints, if something is not correctly configured.

eternalctl operations script

`eternal` offers a handy **operations script** which can be called e.g. to check, whether the service is up, to restart the service, to query how many active sessions exist, to change logging behaviour and so on.

Hint: If `eternalctl` is not part of your `$PATH`, consider either sym-linking it (e.g. `'ln -s /opt/eternal/bin/eternalctl /usr/local/bin/eternalctl'`) or call it from the default `eternal` directory directly: e.g. `/opt/eternal/bin/eternalctl info`

Delegation of incoming federation traffic

In the following documentation, we use the term `server_name` to refer to that setting in your homeserver configuration file. It appears at the ends of user ids, and tells other homeservers where they can find your server.

By default, other homeservers will expect to be able to reach yours via your `server_name`, on port 8448. For example, if you set your `server_name` to `example.com` (so that your user names look like `@user:example.com`), other servers will try to connect to yours at `https://example.com:8448/`.

Delegation is a Matrix feature allowing a homeserver admin to retain a `server_name` of `example.com` so that user IDs, room aliases, etc continue to look like `*:example.com`, whilst having federation traffic routed to a different server and/or port (e.g. `synapse.example.com:443`).

.well-known delegation

To use this method, you need to be able to configure the server at `https://<server_name>` to serve a file at `https://<server_name>/.well-known/matrix/server`. There are two ways to do this, shown below.

Note that the `.well-known` file is hosted on the default port for `https` (port 443).

External server

For maximum flexibility, you need to configure an external server such as nginx, Apache or HAProxy to serve the `https://<server_name>/.well-known/matrix/server` file. Setting up such a server is out of the scope of this documentation, but note that it is often possible to configure your [reverse proxy](#) for this.

The URL `https://<server_name>/.well-known/matrix/server` should be configured to return a JSON structure containing the key `m.server` like this:

```
{
  "m.server": "<synapse.server.name>[:<yourport>]"
}
```

In our example (where we want federation traffic to be routed to `https://synapse.example.com`, on port 443), this would mean that `https://example.com/.well-known/matrix/server` should return:

```
{  
  "m.server": "synapse.example.com:443"  
}
```

Note, specifying a port is optional. If no port is specified, then it defaults to 8448.

Serving a `.well-known/matrix/server` file with Synapse

If you are able to set up your domain so that `https://<server_name>` is routed to Synapse (i.e., the only change needed is to direct federation traffic to port 443 instead of port 8448), then it is possible to configure Synapse to serve a suitable `.well-known/matrix/server` file. To do so, add the following to your `homeserver.yaml` file:

```
serve_server_wellknown: true
```

Note: this *only* works if `https://<server_name>` is routed to Synapse, so is generally not suitable if Synapse is hosted at a subdomain such as `https://synapse.example.com`.

SRV DNS record delegation

It is also possible to do delegation using a SRV DNS record. However, that is generally not recommended, as it can be difficult to configure the TLS certificates correctly in this case, and it offers little advantage over `.well-known` delegation.

Please keep in mind that server delegation is a function of server-server communication, and as such using SRV DNS records will not cover use cases involving client-server comms. This means setting global client settings (such as a Jitsi endpoint, or disabling creating new rooms as encrypted by default, etc) will still require that you serve a file from the `https://<server_name>/.well-known/` endpoints defined in the spec! If you are considering using SRV DNS delegation to avoid serving files from this endpoint, consider the impact that you will not be able to change those client-based default values globally, and will be relegated to the featureset of the configuration of each individual client.

However, if you really need it, you can find some documentation on what such a record should look like and how Synapse will use it in [the Matrix specification](#).

Delegation FAQ

When do I need delegation?

If your homeserver's APIs are accessible on the default federation port (8448) and the domain your `server_name` points to, you do not need any delegation.

For instance, if you registered `example.com` and pointed its DNS A record at a fresh server, you could install Synapse on that host, giving it a `server_name` of `example.com`, and once a reverse proxy has been set up to proxy all requests sent to the port `8448` and serve TLS certificates for `example.com`, you wouldn't need any delegation set up.

However, if your homeserver's APIs aren't accessible on port 8448 and on the domain `server_name` points to, you will need to let other servers know how to find it using delegation.

Should I use a reverse proxy for federation traffic?

Generally, using a reverse proxy for both the federation and client traffic is a good idea, since it saves handling TLS traffic in Synapse. See [the reverse proxy documentation](#) for information on setting up a reverse proxy.

Upgrading Synapse

Before upgrading check if any special steps are required to upgrade from the version you currently have installed to the current version of Synapse. The extra instructions that may be required are listed later in this document.

- Check that your versions of Python and PostgreSQL are still supported.

Synapse follows upstream lifecycles for [Python](#) and [PostgreSQL](#), and removes support for versions which are no longer maintained.

The website <https://endoflife.date> also offers convenient summaries.

- If Synapse was installed using [prebuilt packages](#), you will need to follow the normal process for upgrading those packages.
- If Synapse was installed using pip then upgrade to the latest version by running:

```
pip install --upgrade matrix-synapse
```

- If Synapse was installed from source, then:

1. Obtain the latest version of the source code. Git users can run `git pull` to do this.

2. If you're running Synapse in a virtualenv, make sure to activate it before upgrading. For example, if Synapse is installed in a virtualenv in `~/synapse/env` then run:

```
source ~/synapse/env/bin/activate
pip install --upgrade .
```

Include any relevant extras between square brackets, e.g. `pip install --upgrade ".[postgres,oidc]"`.

3. If you're using `poetry` to manage a Synapse installation, run:

```
poetry install
```

Include any relevant extras with `--extras`, e.g. `poetry install --extras postgres --extras oidc`. It's probably easiest to run `poetry install --extras all`.

4. Restart Synapse:

```
synctl restart
```

To check whether your update was successful, you can check the running server version with:

```
# you may need to replace 'localhost:8008' if synapse is not configured
# to listen on port 8008.

curl http://localhost:8008/_synapse/admin/v1/server_version
```

Rolling back to older versions

Rolling back to previous releases can be difficult, due to database schema changes between releases. Where we have been able to test the rollback process, this will be noted below.

In general, you will need to undo any changes made during the upgrade process, for example:

- pip:

```
source env/bin/activate
# replace `1.3.0` accordingly:
pip install matrix-synapse==1.3.0
```

- Debian:

```
# replace `1.3.0` and `stretch` accordingly:
wget https://packages.matrix.org/debian/pool/main/m/matrix-synapse-
matrix-synapse-py3_1.3.0+stretch1_amd64.deb
dpkg -i matrix-synapse-py3_1.3.0+stretch1_amd64.deb
```

Generally Synapse database schemas are compatible across multiple versions, but once a version of Synapse is deployed you may not be able to roll back automatically. The following table gives the version ranges and the earliest version they can be rolled back to. E.g. Synapse versions v1.58.0 through v1.61.1 can be rolled back safely to v1.57.0, but starting with v1.62.0 it is only safe to roll back to v1.61.0.

Versions	Compatible version
v1.0.0 – v1.2.1	v1.0.0
v1.3.0 – v1.8.0	v1.3.0
v1.9.0 – v1.12.4	v1.9.0
v1.13.0 – v1.25.0	v1.13.0
v1.26.0 – v1.44.0	v1.26.0

Versions	Compatible version
v1.45.0 – v1.47.1	v1.38.0
v1.48.0 – v1.51.0	v1.39.0
v1.52.0 – v1.57.1	v1.49.0
v1.58.0 – v1.61.1	v1.57.0
v1.62.0 – v1.63.1	v1.61.0
v1.64.0 – v1.69.0	v1.62.0
v1.70.0 – v1.82.0	v1.68.0
v1.83.0 – v1.84.1	v1.77.0
v1.85.0 – v1.91.2	v1.83.0
v1.92.0 – v1.97.0	v1.90.0
v1.98.0 – v1.105.0	v1.96.0
v1.105.1 – v1.125.0	v1.100.0

Upgrading from a very old version

You need to read all of the upgrade notes for each version between your current version and the latest so that you can update your dependencies, environment, config files, etc. if necessary. But you do not need to perform an upgrade to each individual version that was missed.

We do not have a list of which versions must be installed. Instead, we recommend that you upgrade through each incompatible database schema version, which would give you the ability to roll back the maximum number of versions should anything go wrong. See [Rolling back to older versions](#) above.

Additionally, new versions of Synapse will occasionally run database migrations and background updates to update the database. Synapse will not start until database migrations are complete. You should wait until background updates from each upgrade are complete before moving on to the next upgrade, to avoid stacking them up. You can monitor the currently running background updates with [the Admin API](#).

Upgrading to v1.122.0

Dropping support for PostgreSQL 11 and 12

In line with our [deprecation policy](#), we've dropped support for PostgreSQL 11 and 12, as they are no longer supported upstream. This release of Synapse requires PostgreSQL 13+.

Upgrading to v1.120.0

Removal of experimental MSC3886 feature

MSC3886 has been closed (and will not enter the Matrix spec). As such, we are removing the experimental support for it in this release.

The `experimental_features.msc3886_endpoint` configuration option has been removed.

Authenticated media is now enforced by default

The `enable_authenticated_media` configuration option now defaults to true.

This means that clients and remote (federated) homeservers now need to use the authenticated media endpoints in order to download media from your homeserver.

As an exception, existing media that was stored on the server prior to this option changing to `true` will still be accessible over the unauthenticated endpoints.

The matrix.org homeserver has already been running with this option enabled since September 2024, so most common clients and homeservers should already be compatible.

With that said, administrators who wish to disable this feature for broader compatibility can still do so by manually configuring `enable_authenticated_media: False`.

Upgrading to v1.119.0

Minimum supported Python version

The minimum supported Python version has been increased from v3.8 to v3.9. You will need Python 3.9+ to run Synapse v1.119.0 (due out Nov 7th, 2024).

If you use current versions of the Matrix.org-distributed Docker images, no action is required. Please note that support for Ubuntu `focal` was dropped as well since it uses Python 3.8.

Upgrading to v1.111.0

New worker endpoints for authenticated client and federation media

Media repository workers handling Media APIs can now handle the following endpoint patterns:

```
^/_matrix/client/v1/media/.*$  
^/_matrix/federation/v1/media/.*$
```

Please update your reverse proxy configuration.

Upgrading to v1.106.0

Minimum supported Rust version

The minimum supported Rust version has been increased from v1.65.0 to v1.66.0. Users building from source will need to ensure their `rustc` version is up to date.

Upgrading to v1.100.0

Minimum supported Rust version

The minimum supported Rust version has been increased from v1.61.0 to v1.65.0. Users building from source will need to ensure their `rustc` version is up to date.

Upgrading to v1.93.0

Minimum supported Rust version

The minimum supported Rust version has been increased from v1.60.0 to v1.61.0. Users building from source will need to ensure their `rustc` version is up to date.

Upgrading to v1.90.0

App service query parameter authorization is now a configuration option

Synapse v1.81.0 deprecated application service authorization via query parameters as this is considered insecure - and from Synapse v1.71.0 forwards the application service token has also been sent via the `Authorization` header], making the insecure query parameter authorization redundant. Since removing the ability to continue to use query parameters could break backwards compatibility it has now been put behind a configuration option, `use_appservice_legacy_authorization`. This option defaults to false, but can be activated by adding

```
use_appservice_legacy_authorization: true
```

to your configuration.

Upgrading to v1.89.0

Removal of unspeced `user` property for `/register`

Application services can no longer call `/register` with a `user` property to create new users. The standard `username` property should be used instead. See the [Application Service specification](#) for more information.

Upgrading to v1.88.0

Minimum supported Python version

The minimum supported Python version has been increased from v3.7 to v3.8. You will need Python 3.8 to run Synapse v1.88.0 (due out July 18th, 2023).

If you use current versions of the Matrix.org-distributed Debian packages or Docker images, no action is required.

Removal of `worker_replication_*` settings

As mentioned previously in [Upgrading to v1.84.0](#), the following deprecated settings are being removed in this release of Synapse:

- `worker_replication_host`
- `worker_replication_http_port`
- `worker_replication_http_tls`

Please ensure that you have migrated to using `main` on your shared configuration's `instance_map` (or create one if necessary). This is required if you have *any* workers at all; administrators of single-process (monolith) installations don't need to do anything.

For an illustrative example, please see [Upgrading to v1.84.0](#) below.

Upgrading to v1.86.0

Minimum supported Rust version

The minimum supported Rust version has been increased from v1.58.1 to v1.60.0. Users building from source will need to ensure their `rustc` version is up to date.

Upgrading to v1.85.0

Application service registration with "user" property deprecation

Application services should ensure they call the `/register` endpoint with a `username` property. The legacy `user` property is considered deprecated and should no longer be included.

A future version of Synapse (v1.88.0 or later) will remove support for legacy application service login.

Upgrading to v1.84.0

Deprecation of `worker_replication_*` configuration settings

When using workers,

- `worker_replication_host`
- `worker_replication_http_port`
- `worker_replication_http_tls`

should now be removed from individual worker YAML configurations and the main process should instead be added to the `instance_map` in the shared YAML configuration, using the name `main`.

The old `worker_replication_*` settings are now considered deprecated and are expected to be removed in Synapse v1.88.0.

Example change

Before:

Shared YAML

```
instance_map:  
  generic_worker1:  
    host: localhost  
    port: 5678  
    tls: false
```

Worker YAML

```

worker_app: synapse.app.generic_worker
worker_name: generic_worker1

worker_replication_host: localhost
worker_replication_http_port: 3456
worker_replication_http_tls: false

worker_listeners:
- type: http
  port: 1234
  resources:
    - names: [client, federation]
- type: http
  port: 5678
  resources:
    - names: [replication]

worker_log_config: /etc/matrix-synapse/generic-worker-log.yaml

```

After:

Shared YAML

```

instance_map:
  main:
    host: localhost
    port: 3456
    tls: false
  generic_worker1:
    host: localhost
    port: 5678
    tls: false

```

Worker YAML

```

worker_app: synapse.app.generic_worker
worker_name: generic_worker1

worker_listeners:
- type: http
  port: 1234
  resources:
    - names: [client, federation]
- type: http
  port: 5678
  resources:
    - names: [replication]

worker_log_config: /etc/matrix-synapse/generic-worker-log.yaml

```

Notes:

- `tls` is optional but mirrors the functionality of `worker_replication_http_tls`

Upgrading to v1.81.0

Application service path & authentication deprecations

Synapse now attempts the versioned appservice paths before falling back to the [legacy paths](#). Usage of the legacy routes should be considered deprecated.

Additionally, Synapse has supported sending the application service access token via [the Authorization header](#) since v1.70.0. For backwards compatibility it is *also* sent as the `access_token` query parameter. This is insecure and should be considered deprecated.

A future version of Synapse (v1.88.0 or later) will remove support for legacy application service routes and query parameter authorization.

Upgrading to v1.80.0

Reporting events error code change

Before this update, the [POST /_matrix/client/v3/rooms/{roomId}/report/{eventId}](#) endpoint would return a `403` if a user attempted to report an event that they did not have access to. This endpoint will now return a `404` in this case instead.

Clients that implement event reporting should check that their error handling code will handle this change.

Upgrading to v1.79.0

The `on_threepid_bind` module callback method has been deprecated

Synapse v1.79.0 deprecates the `on_threepid_bind` "third-party rules" Synapse module callback method in favour of a new module method,

[on_add_user_third_party_identifier](#). `on_threepid_bind` will be removed in a future version of Synapse. You should check whether any Synapse modules in use in your deployment are making use of `on_threepid_bind`, and update them where possible.

The arguments and functionality of the new method are the same.

The justification behind the name change is that the old method's name, `on_threepid_bind`, was misleading. A user is considered to "bind" their third-party ID to their Matrix ID only if they do so via an [identity server](#) (so that users on other homeservers may find them). But this method was not called in that case - it was only called when a user added a third-party identifier on the local homeserver.

Module developers may also be interested in the related `on_remove_user_third_party_identifier` module callback method that was also added in Synapse v1.79.0. This new method is called when a user removes a third-party identifier from their account.

Upgrading to v1.78.0

Deprecate the

`/_synapse/admin/v1/media/<server_name>/delete`
admin API

Synapse 1.78.0 replaces the `/_synapse/admin/v1/media/<server_name>/delete` admin API with an identical endpoint at `/_synapse/admin/v1/media/delete`. Please update your tooling to use the new endpoint. The deprecated version will be removed in a future release.

Upgrading to v1.76.0

Faster joins are enabled by default

When joining a room for the first time, Synapse 1.76.0 will request a partial join from the other server by default. Previously, server admins had to opt-in to this using an experimental config flag.

Server admins can opt out of this feature for the time being by setting

```
experimental:  
  faster_joins: false
```

in their server config.

Changes to the account data replication streams

Synapse has changed the format of the account data and devices replication streams (between workers). This is a forwards- and backwards-incompatible change: v1.75 workers cannot process account data replicated by v1.76 workers, and vice versa.

Once all workers are upgraded to v1.76 (or downgraded to v1.75), account data and device replication will resume as normal.

Minimum version of Poetry is now 1.3.2

The minimum supported version of Poetry is now 1.3.2 (previously 1.2.0, [since Synapse 1.67](#)). If you have used `poetry install` to install Synapse from a source checkout, you should upgrade poetry: see its [installation instructions](#). For all other installation methods, no action is required.

Upgrading to v1.74.0

Unicode support in user search

This version introduces optional support for an [improved user search dealing with Unicode characters](#).

If you want to take advantage of this feature you need to install PyICU, the ICU native dependency and its development headers so that PyICU can build since no prebuilt wheels are available.

You can follow [the PyICU documentation](#) to do so, and then do `pip install matrix-synapse[user-search]` for a PyPI install.

Docker images and Debian packages need nothing specific as they already include or specify ICU as an explicit dependency.

User directory rebuild

Synapse 1.74 queues a background update [to rebuild the user directory](#), in order to fix missing or erroneous entries.

When this update begins, the user directory will be cleared out and rebuilt from scratch. User directory lookups will be incomplete until the rebuild completes. Admins can monitor the rebuild's progress by using the [Background update Admin API](#).

Upgrading to v1.73.0

Legacy Prometheus metric names have now been removed

Synapse v1.69.0 included the deprecation of legacy Prometheus metric names and offered an option to disable them. Synapse v1.71.0 disabled legacy Prometheus metric names by default.

This version, v1.73.0, removes those legacy Prometheus metric names entirely. This also means that the `enable_legacy_metrics` configuration option has been removed; it will no longer be possible to re-enable the legacy metric names.

If you use metrics and have not yet updated your Grafana dashboard(s), Prometheus console(s) or alerting rule(s), please consider doing so when upgrading to this version. Note that the included Grafana dashboard was updated in v1.72.0 to correct some metric names which were missed when legacy metrics were disabled by default.

See [v1.69.0: Deprecation of legacy Prometheus metric names](#) for more context.

Upgrading to v1.72.0

Dropping support for PostgreSQL 10

In line with our [deprecation policy](#), we've dropped support for PostgreSQL 10, as it is no longer supported upstream.

This release of Synapse requires PostgreSQL 11+.

Upgrading to v1.71.0

Removal of the `generate_short_term_login_token` module API method

As announced with the release of [Synapse 1.69.0](#), the deprecated `generate_short_term_login_token` module method has been removed.

Modules relying on it can instead use the `create_login_token` method.

Changes to the events received by application services (interest)

To align with spec (changed in [MSC3905](#)), Synapse now only considers local users to be interesting. In other words, the `users` namespace regex is only be applied against local users of the homeserver.

Please note, this probably doesn't affect the expected behavior of your application service, since an interesting local user in a room still means all messages in the room (from local or remote users) will still be considered interesting. And matching a room with the `rooms` or `aliases` namespace regex will still consider all events sent in the room to be interesting to the application service.

If one of your application service's `users` regex was intending to match a remote user, this will no longer match as you expect. The behavioral mismatch between matching all local users and some remote users is why the spec was changed/clarified and this caveat is no longer supported.

Legacy Prometheus metric names are now disabled by default

Synapse v1.71.0 disables legacy Prometheus metric names by default. For administrators that still rely on them and have not yet had chance to update their uses of the metrics, it's still possible to specify `enable_legacy_metrics: true` in the configuration to re-enable them temporarily.

Synapse v1.73.0 will **remove legacy metric names altogether** and at that point, it will no longer be possible to re-enable them.

If you do not use metrics or you have already updated your Grafana dashboard(s), Prometheus console(s) and alerting rule(s), there is no action needed.

See [v1.69.0: Deprecation of legacy Prometheus metric names](#).

Upgrading to v1.69.0

Changes to the receipts replication streams

Synapse now includes information indicating if a receipt applies to a thread when replicating it to other workers. This is a forwards- and backwards-incompatible change: v1.68 and workers cannot process receipts replicated by v1.69 workers, and vice versa.

Once all workers are upgraded to v1.69 (or downgraded to v1.68), receipts replication will resume as normal.

Deprecation of legacy Prometheus metric names

In current versions of Synapse, some Prometheus metrics are emitted under two different names, with one of the names being older but non-compliant with OpenMetrics and Prometheus conventions and one of the names being newer but compliant.

Synapse v1.71.0 will turn the old metric names off *by default*. For administrators that still rely on them and have not had chance to update their uses of the metrics, it's possible to specify `enable_legacy_metrics: true` in the configuration to re-enable them temporarily.

Synapse v1.73.0 will **remove legacy metric names altogether** and it will no longer be possible to re-enable them.

The Grafana dashboard, Prometheus recording rules and Prometheus Consoles included in the `contrib` directory in the Synapse repository have been updated to no longer rely on the legacy names. These can be used on a current version of Synapse because current versions of Synapse emit both old and new names.

You may need to update your alerting rules or any other rules that depend on the names of Prometheus metrics. If you want to test your changes before legacy names are disabled by default, you may specify `enable_legacy_metrics: false` in your homeserver configuration.

A list of affected metrics is available on the [Metrics How-to page](#).

Deprecation of the `generate_short_term_login_token` module API method

The following method of the module API has been deprecated, and is scheduled to be removed in v1.71.0:

```
def generate_short_term_login_token(
    self,
    user_id: str,
    duration_in_ms: int = (2 * 60 * 1000),
    auth_provider_id: str = "",
    auth_provider_session_id: Optional[str] = None,
) -> str:
    ...
```

It has been replaced by an asynchronous equivalent:

```
async def create_login_token(
    self,
    user_id: str,
    duration_in_ms: int = (2 * 60 * 1000),
    auth_provider_id: Optional[str] = None,
    auth_provider_session_id: Optional[str] = None,
) -> str:
    ...
```

Synapse will log a warning when a module uses the deprecated method, to help administrators find modules using it.

Upgrading to v1.68.0

Two changes announced in the upgrade notes for v1.67.0 have now landed in v1.68.0.

SQLite version requirement

Synapse now requires a SQLite version of 3.27.0 or higher if SQLite is configured as Synapse's database.

Installations using

- Docker images from matrix.org,
- Debian packages from [Matrix.org](https://matrix.org), or
- a PostgreSQL database

are not affected.

Rust requirement when building from source.

Building from a source checkout of Synapse now requires a recent Rust compiler (currently Rust 1.58.1, but see also the [Platform Dependency Policy](#)).

Installations using

- Docker images from [matrixdotorg](#),
- Debian packages from [Matrix.org](#), or
- PyPI wheels via `pip install matrix-synapse` (on supported platforms and architectures)

will not be affected.

Upgrading to v1.67.0

Direct TCP replication is no longer supported: migrate to Redis

Redis support was added in v1.13.0 with it becoming the recommended method in v1.18.0. It replaced the old direct TCP connections (which was deprecated as of v1.18.0) to the main process. With Redis, rather than all the workers connecting to the main process, all the workers and the main process connect to Redis, which relays replication commands between processes. This can give a significant CPU saving on the main process and is a prerequisite for upcoming performance improvements.

To migrate to Redis add the `redis` config, and remove the TCP `replication` listener from config of the master and `worker_replication_port` from worker config. Note that a HTTP listener with a `replication` resource is still required.

Minimum version of Poetry is now v1.2.0

The minimum supported version of poetry is now 1.2. This should only affect those installing from a source checkout.

Rust requirement in the next release

From the next major release (v1.68.0) installing Synapse from a source checkout will require a recent Rust compiler. Those using packages or `pip install matrix-synapse` will not be

affected.

The simplest way of installing Rust is via [rustup.rs](#)

SQLite version requirement in the next release

From the next major release (v1.68.0) Synapse will require SQLite 3.27.0 or higher. Synapse v1.67.0 will be the last major release supporting SQLite versions 3.22 to 3.26.

Those using Docker images or Debian packages from Matrix.org will not be affected. If you have installed from source, you should check the version of SQLite used by Python with:

```
python -c "import sqlite3; print(sqlite3.sqlite_version)"
```

If this is too old, refer to your distribution for advice on upgrading.

Upgrading to v1.66.0

Delegation of email validation no longer supported

As of this version, Synapse no longer allows the tasks of verifying email address ownership, and password reset confirmation, to be delegated to an identity server. This removal was previously planned for Synapse 1.64.0, but was [delayed](#) until now to give homeserver administrators more notice of the change.

To continue to allow users to add email addresses to their homeserver accounts, and perform password resets, make sure that Synapse is configured with a working email server in the `email` configuration section (including, at a minimum, a `notif_from` setting.)

Specifying an `email` setting under `account_threepid_delegates` will now cause an error at startup.

Upgrading to v1.64.0

Deprecation of the ability to delegate e-mail verification to identity servers

Synapse v1.66.0 will remove the ability to delegate the tasks of verifying email address ownership, and password reset confirmation, to an identity server.

If you require your homeserver to verify e-mail addresses or to support password resets via e-mail, please configure your homeserver with SMTP access so that it can send e-mails on its own behalf. [Consult the configuration documentation for more information.](#)

The option that will be removed is `account_threepid_delegates.email`.

Changes to the event replication streams

Synapse now includes a flag indicating if an event is an outlier when replicating it to other workers. This is a forwards- and backwards-incompatible change: v1.63 and workers cannot process events replicated by v1.64 workers, and vice versa.

Once all workers are upgraded to v1.64 (or downgraded to v1.63), event replication will resume as normal.

frozendict release

[frozendict 2.3.3](#) has recently been released, which fixes a memory leak that occurs during `/sync` requests. We advise server administrators who installed Synapse via pip to upgrade frozendict with `pip install --upgrade frozendict`. The Docker image `matrixdotorg/synapse` and the Debian packages from `packages.matrix.org` already include the updated library.

Upgrading to v1.62.0

New signatures for spam checker callbacks

As a followup to changes in v1.60.0, the following spam-checker callbacks have changed signature:

- `user_may_join_room`
- `user_may_invite`
- `user_may_send_3pid_invite`
- `user_may_create_room`
- `user_may_create_room_alias`
- `user_may_publish_room`
- `check_media_file_for_spam`

For each of these methods, the previous callback signature has been deprecated.

Whereas callbacks used to return `bool`, they should now return

```
Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes"].
```

For instance, if your module implements `user_may_join_room` as follows:

```
async def user_may_join_room(self, user_id: str, room_id: str, is_invited: bool):
    if ...:
        # Request is spam
        return False
    # Request is not spam
    return True
```

you should rewrite it as follows:

```
async def user_may_join_room(self, user_id: str, room_id: str, is_invited: bool):
    if ...:
        # Request is spam, mark it as forbidden (you may use some more precise
        # error
        # code if it is useful).
        return synapse.module_api.errors.Codes.FORBIDDEN
    # Request is not spam, mark it as such.
    return synapse.module_api.NOT_SPAM
```

Upgrading to v1.61.0

Removal of deprecated community/groups

This release of Synapse will remove deprecated community/groups from codebase.

Worker endpoints

For those who have deployed workers, following worker endpoints will no longer exist and they can be removed from the reverse proxy configuration:

- `^/_matrix/federation/v1/get_groups_publicised$`
- `^/_matrix/client/(r0|v3|unstable)/joined_groups$`
- `^/_matrix/client/(r0|v3|unstable)/publicised_groups$`
- `^/_matrix/client/(r0|v3|unstable)/publicised_groups/`
- `^/_matrix/federation/v1/groups/`
- `^/_matrix/client/(r0|v3|unstable)/groups/`

Upgrading to v1.60.0

Adding a new unique index to `state_group_edges` could fail if your database is corrupted

This release of Synapse will add a unique index to the `state_group_edges` table, in order to prevent accidentally introducing duplicate information (for example, because a database backup was restored multiple times).

Duplicate rows being present in this table could cause drastic performance problems; see [issue 11779](#) for more details.

If your Synapse database already has had duplicate rows introduced into this table, this could fail, with either of these errors:

On Postgres:

```
synapse.storage.background_updates - 623 - INFO - background_updates-0 - Adding
index state_group_edges_unique_idx to state_group_edges
synapse.storage.background_updates - 282 - ERROR - background_updates-0 - Error
doing update
...
psycopg2.errors.UniqueViolation: could not create unique index
"state_group_edges_unique_idx"
DETAIL: Key (state_group, prev_state_group)=(2, 1) is duplicated.
```

(The numbers may be different.)

On SQLite:

```
synapse.storage.background_updates - 623 - INFO - background_updates-0 - Adding
index state_group_edges_unique_idx to state_group_edges
synapse.storage.background_updates - 282 - ERROR - background_updates-0 - Error
doing update
...
sqlite3.IntegrityError: UNIQUE constraint failed: state_group_edges.state_group,
state_group_edges.prev_state_group
```

► Expand this section for steps to resolve this problem

New signature for the spam checker callback `check_event_for_spam`

The previous signature has been deprecated.

Whereas `check_event_for_spam` callbacks used to return `Union[str, bool]`, they should now return `Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes"]`.

This is part of an ongoing refactoring of the SpamChecker API to make it less ambiguous and more powerful.

If your module implements `check_event_for_spam` as follows:

```
async def check_event_for_spam(event):
    if ...:
        # Event is spam
        return True
    # Event is not spam
    return False
```

you should rewrite it as follows:

```
async def check_event_for_spam(event):
    if ...:
        # Event is spam, mark it as forbidden (you may use some more precise
        # error
        # code if it is useful).
        return synapse.module_api.errors.Codes.FORBIDDEN
    # Event is not spam, mark it as such.
    return synapse.module_api.NOT_SPAM
```

Upgrading to v1.59.0

Device name lookup over federation has been disabled by default

The names of user devices are no longer visible to users on other homeservers by default. Device IDs are unaffected, as these are necessary to facilitate end-to-end encryption.

To re-enable this functionality, set the `allow_device_name_lookup_over_federation` homeserver config option to `true`.

Deprecation of the `synapse.app.appservice` and `synapse.app.user_dir` worker application types

The `synapse.app.appservice` worker application type allowed you to configure a single worker to use to notify application services of new events, as long as this functionality was

disabled on the main process with `notify_appservices: False`. Further, the `synapse.app.user_dir` worker application type allowed you to configure a single worker to be responsible for updating the user directory, as long as this was disabled on the main process with `update_user_directory: False`.

To unify Synapse's worker types, the `synapse.app.appservice` worker application type and the `notify_appservices` configuration option have been deprecated. The `synapse.app.user_dir` worker application type and `update_user_directory` configuration option have also been deprecated.

To get the same functionality as was provided by the deprecated options, it's now recommended that the `synapse.app.generic_worker` worker application type is used and that the `notify_appservices_from_worker` and/or `update_user_directory_from_worker` options are set to the name of a worker.

For the time being, the old options can be used alongside the new options to make it easier to transition between the two configurations, however please note that:

- the options must not contradict each other (otherwise Synapse won't start); and
- the `notify_appservices` and `update_user_directory` options will be removed in a future release of Synapse.

Please see the [Notifying Application Services](#) and [Updating the User Directory](#) sections of the worker documentation for more information.

Upgrading to v1.58.0

Groups/communities feature has been disabled by default

The non-standard groups/communities feature in Synapse has been disabled by default and will be removed in Synapse v1.61.0.

Upgrading to v1.57.0

Changes to database schema for application services

Synapse v1.57.0 includes a [change](#) to the way transaction IDs are managed for application services. If your deployment uses a dedicated worker for application service traffic, **it must be stopped** when the database is upgraded (which normally happens when the main

process is upgraded), to ensure the change is made safely without any risk of reusing transaction IDs.

Deployments which do not use separate worker processes can be upgraded as normal. Similarly, deployments where no application services are in use can be upgraded as normal.

► Recovering from an incorrect upgrade

Upgrading to v1.56.0

Open registration without verification is now disabled by default

Synapse will refuse to start if registration is enabled without email, captcha, or token-based verification unless the new config flag `enable_registration_without_verification` is set to "true".

Groups/communities feature has been deprecated

The non-standard groups/communities feature in Synapse has been deprecated and will be disabled by default in Synapse v1.58.0.

You can test disabling it by adding the following to your homeserver configuration:

```
experimental_features:  
  groups_enabled: false
```

Change in behaviour for PostgreSQL databases with unsafe locale

Synapse now refuses to start when using PostgreSQL with non-`C` values for `COLLATE` and `CTYPE` unless the config flag `allow_unsafe_locale`, found in the database section of the configuration file, is set to `true`. See the [PostgreSQL documentation](#) for more information and instructions on how to fix a database with incorrect values.

Upgrading to v1.55.0

synctl script has been moved

The `synctl` script [has been made](#) an [entry point](#) and no longer exists at the root of Synapse's source tree. If you wish to use `synctl` to manage your homeserver, you should invoke `synctl` directly, e.g. `synctl start` instead of `./synctl start` or `/path/to/synctl start`.

You will need to ensure `synctl` is on your `PATH`.

- This is automatically the case when using [Debian packages](#) or [docker images](#) provided by Matrix.org.
- When installing from a wheel, sdist, or PyPI, a `synctl` executable is added to your Python installation's `bin`. This should be on your `PATH` automatically, though you might need to activate a virtual environment depending on how you installed Synapse.

Compatibility dropped for Mjolnir 1.3.1 and earlier

Synapse v1.55.0 drops support for Mjolnir 1.3.1 and earlier. If you use the Mjolnir module to moderate your homeserver, please upgrade Mjolnir to version 1.3.2 or later before upgrading Synapse.

Upgrading to v1.54.0

Legacy structured logging configuration removal

This release removes support for the `structured: true` logging configuration which was deprecated in Synapse v1.23.0. If your logging configuration contains `structured: true` then it should be modified based on the [structured logging documentation](#).

Upgrading to v1.53.0

Dropping support for `webclient` listeners and non-HTTP(S) `web_client_location`

Per the deprecation notice in Synapse v1.51.0, listeners of type `webclient` are no longer supported and configuring them is now a configuration error.

Configuring a non-HTTP(S) `web_client_location` configuration is now a configuration error. Since the `webclient` listener is no longer supported, this setting only applies to the root path `/` of Synapse's web server and no longer the `/_matrix/client/` path.

Stabilisation of MSC3231

The unstable validity-check endpoint for the [Registration Tokens](#) feature has been stabilised and moved from:

`/_matrix/client/unstable/org.matrix.msc3231/register/org.matrix.msc3231.login.registration_token/validity`

to:

`/_matrix/client/v1/register/m.login.registration_token/validity`

Please update any relevant reverse proxy or firewall configurations appropriately.

Time-based cache expiry is now enabled by default

Formerly, entries in the cache were not evicted regardless of whether they were accessed after storing. This behavior has now changed. By default entries in the cache are now evicted after 30m of not being accessed. To change the default behavior, go to the `caches` section of the config and change the `expire_caches` and `cache_entry_ttl` flags as necessary. Please note that these flags replace the `expiry_time` flag in the config. The `expiry_time` flag will still continue to work, but it has been deprecated and will be removed in the future.

Deprecation of capability `org.matrix.msc3283.*`

The `capabilities` of MSC3283 from the REST API `/_matrix/client/r0/capabilities` becomes stable.

The old `capabilities`

- `org.matrix.msc3283.set_displayname`,
- `org.matrix.msc3283.set_avatar_url` and
- `org.matrix.msc3283.3pid_changes`

are deprecated and scheduled to be removed in Synapse v1.54.0.

The new `capabilities`

- `m.set_displayname`,
- `m.set_avatar_url` and
- `m.3pid_changes`

are now active by default.

Removal of `user_may_create_room_with_invites`

As announced with the release of [Synapse 1.47.0](#), the deprecated `user_may_create_room_with_invites` module callback has been removed.

Modules relying on it can instead implement `user_may_invite` and use the `get_room_state` module API to infer whether the invite is happening while creating a room (see [this function](#) as an example). Alternately, modules can also implement `on_create_room`.

Upgrading to v1.52.0

Twisted security release

Note that [Twisted 22.1.0](#) has recently been released, which fixes a [security issue](#) within the Twisted library. We do not believe Synapse is affected by this vulnerability, though we advise server administrators who installed Synapse via pip to upgrade Twisted with `pip install --upgrade Twisted treq` as a matter of good practice. The Docker image `matrixdotorg/synapse` and the Debian packages from `packages.matrix.org` are using the updated library.

Upgrading to v1.51.0

Deprecation of `webclient` listeners and non-HTTP(S) `web_client_location`

Listeners of type `webclient` are deprecated and scheduled to be removed in Synapse v1.53.0.

Similarly, a non-HTTP(S) `web_client_location` configuration is deprecated and will become a configuration error in Synapse v1.53.0.

Upgrading to v1.50.0

Dropping support for old Python and Postgres versions

In line with our [deprecation policy](#), we've dropped support for Python 3.6 and PostgreSQL 9.6, as they are no longer supported upstream.

This release of Synapse requires Python 3.7+ and PostgreSQL 10+.

Upgrading to v1.47.0

Removal of old Room Admin API

The following admin APIs were deprecated in [Synapse 1.34](#) (released on 2021-05-17) and have now been removed:

- `POST /_synapse/admin/v1/<room_id>/delete`

Any scripts still using the above APIs should be converted to use the [Delete Room API](#).

Deprecation of the `user_may_create_room_with_invites` module callback

The `user_may_create_room_with_invites` is deprecated and will be removed in a future version of Synapse. Modules implementing this callback can instead implement `user_may_invite` and use the `get_room_state` module API method to infer whether the invite is happening in the context of creating a room.

We plan to remove this callback in January 2022.

Upgrading to v1.45.0

Changes required to media storage provider modules when reading from the Synapse configuration object

Media storage provider modules that read from the Synapse configuration object (i.e. that read the value of `hs.config.[...]`) now need to specify the configuration section they're reading from. This means that if a module reads the value of e.g.

`hs.config.media_store_path`, it needs to replace it with
`hs.config.media.media_store_path`.

Upgrading to v1.44.0

The URL preview cache is no longer mirrored to storage providers

The `url_cache/` and `url_cache_thumbnails/` directories in the media store are no longer mirrored to storage providers. These two directories can be safely deleted from any configured storage providers to reclaim space.

Upgrading to v1.43.0

The spaces summary APIs can now be handled by workers

The [available worker applications documentation](#) has been updated to reflect that calls to the `/spaces`, `/hierarchy`, and `/summary` endpoints can now be routed to workers for both client API and federation requests.

Upgrading to v1.42.0

Removal of old Room Admin API

The following admin APIs were deprecated in [Synapse 1.25](#) (released on 2021-01-13) and have now been removed:

- `POST /_synapse/admin/v1/purge_room`
- `POST /_synapse/admin/v1/shutdown_room/<room_id>`

Any scripts still using the above APIs should be converted to use the [Delete Room API](#).

User-interactive authentication fallback templates can now display errors

This may affect you if you make use of custom HTML templates for the reCAPTCHA ([synapse/res/templates/recaptcha.html](#)) or terms ([synapse/res/templates/terms.html](#)) fallback pages.

The template is now provided an `error` variable if the authentication process failed. See the default templates linked above for an example.

Removal of out-of-date email pushers

Users will stop receiving message updates via email for addresses that were once, but not still, linked to their account.

Upgrading to v1.41.0

Add support for routing outbound HTTP requests via a proxy for federation

Since Synapse 1.6.0 (2019-11-26) you can set a proxy for outbound HTTP requests via `http_proxy/https_proxy` environment variables. This proxy was set for:

- push
- url previews
- phone-home stats
- re captcha validation
- CAS auth validation
- OpenID Connect
- Federation (checking public key revocation)

In this version we have added support for outbound requests for:

- Outbound federation
- Downloading remote media
- Fetching public keys of other servers

These requests use the same proxy configuration. If you have a proxy configuration we recommend to verify the configuration. It may be necessary to adjust the `no_proxy` environment variable.

See [using a forward proxy with Synapse documentation](#) for details.

Deprecation of `template_dir`

The `template_dir` settings in the `sso`, `account_validity` and `email` sections of the configuration file are now deprecated. Server admins should use the new `templates.custom_template_directory` setting in the configuration file and use one single custom template directory for all aforementioned features. Template file names remain unchanged. See [the related documentation](#) for more information and examples.

We plan to remove support for these settings in October 2021.

`/_synapse/admin/v1/users/{userId}/media` must be handled by media workers

The [media repository worker documentation](#) has been updated to reflect that calls to `/_synapse/admin/v1/users/{userId}/media` must now be handled by media repository workers. This is due to the new `DELETE` method of this endpoint modifying the media store.

Upgrading to v1.39.0

Deprecation of the current third-party rules module interface

The current third-party rules module interface is deprecated in favour of the new generic modules system introduced in Synapse v1.37.0. Authors of third-party rules modules can refer to [this documentation](#) to update their modules. Synapse administrators can refer to [this documentation](#) to update their configuration once the modules they are using have been updated.

We plan to remove support for the current third-party rules interface in September 2021.

Upgrading to v1.38.0

Re-indexing of `events` table on Postgres databases

This release includes a database schema update which requires re-indexing one of the larger tables in the database, `events`. This could result in increased disk I/O for several hours or days after upgrading while the migration completes. Furthermore, because we have to keep the old indexes until the new indexes are ready, it could result in a significant, temporary, increase in disk space.

To get a rough idea of the disk space required, check the current size of one of the indexes. For example, from a `psql` shell, run the following sql:

```
SELECT pg_size_pretty(pg_relation_size('events_order_room'));
```

We need to rebuild **four** indexes, so you will need to multiply this result by four to give an estimate of the disk space required. For example, on one particular server:

```
synapse=# select pg_size.pretty(pg_relation_size('events_order_room'));  
pg_size.pretty  
-----  
288 MB  
(1 row)
```

On this server, it would be wise to ensure that at least 1152MB are free.

The additional disk space will be freed once the migration completes.

SQLite databases are unaffected by this change.

Upgrading to v1.37.0

Deprecation of the current spam checker interface

The current spam checker interface is deprecated in favour of a new generic modules system. Authors of spam checker modules can refer to [\[this documentation\]](#) (modules/porting_legacy_module.md to update their modules. Synapse administrators can refer to [this documentation](#) to update their configuration once the modules they are using have been updated.

We plan to remove support for the current spam checker interface in August 2021.

More module interfaces will be ported over to this new generic system in future versions of Synapse.

Upgrading to v1.34.0

room_invite_state_types configuration setting

The `room_invite_state_types` configuration setting has been deprecated and replaced with `room_prejoin_state`. See the [sample configuration file](#).

If you have set `room_invite_state_types` to the default value you should simply remove it from your configuration file. The default value used to be:

```
room_invite_state_types:  
- "m.room.join_rules"  
- "m.room.canonical_alias"  
- "m.room.avatar"  
- "m.room.encryption"  
- "m.room.name"
```

If you have customised this value, you should remove `room_invite_state_types` and configure `room_prejoin_state` instead.

Upgrading to v1.33.0

Account Validity HTML templates can now display a user's expiration date

This may affect you if you have enabled the account validity feature, and have made use of a custom HTML template specified by the `account_validity.template_dir` or `account_validity.account_renewed_html_path` Synapse config options.

The template can now accept an `expiration_ts` variable, which represents the unix timestamp in milliseconds for the future date of which their account has been renewed until. See the [default template](#) for an example of usage.

Also note that a new HTML template, `account_previously_renewed.html`, has been added. This is shown to users when they attempt to renew their account with a valid renewal token that has already been used before. The default template contents can be found [here](#), and can also accept an `expiration_ts` variable. This template replaces the error message users would previously see upon attempting to use a valid renewal token more than once.

Upgrading to v1.32.0

Regression causing connected Prometheus instances to become overwhelmed

This release introduces a [regression](#) that can overwhelm connected Prometheus instances. This issue is not present in Synapse v1.32.0rc1.

If you have been affected, please downgrade to 1.31.0. You then may need to remove excess writeahead logs in order for Prometheus to recover. Instructions for doing so are provided [here](#).

Dropping support for old Python, Postgres and SQLite versions

In line with our [deprecation policy](#), we've dropped support for Python 3.5 and PostgreSQL 9.5, as they are no longer supported upstream.

This release of Synapse requires Python 3.6+ and PostgreSQL 9.6+ or SQLite 3.22+.

Removal of old List Accounts Admin API

The deprecated v1 "list accounts" admin API (`GET /_synapse/admin/v1/users/<user_id>`) has been removed in this version.

The [v2 list accounts API](#) has been available since Synapse 1.7.0 (2019-12-13), and is accessible under `GET /_synapse/admin/v2/users`.

The deprecation of the old endpoint was announced with Synapse 1.28.0 (released on 2021-02-25).

Application Services must use type `m.login.application_service` when registering users

In compliance with the [Application Service spec](#), Application Services are now required to use the `m.login.application_service` type when registering users via the `/_matrix/client/r0/register` endpoint. This behaviour was deprecated in Synapse v1.30.0.

Please ensure your Application Services are up to date.

Upgrading to v1.29.0

Requirement for X-Forwarded-Proto header

When using Synapse with a reverse proxy (in particular, when using the `x_forwarded` option on an HTTP listener), Synapse now expects to receive an `X-Forwarded-Proto` header on incoming HTTP requests. If it is not set, Synapse will log a warning on each received request.

To avoid the warning, administrators using a reverse proxy should ensure that the reverse proxy sets `X-Forwarded-Proto` header to `https` or `http` to indicate the protocol used by the client.

Synapse also requires the `Host` header to be preserved.

See the [reverse proxy documentation](#), where the example configurations have been updated to show how to set these headers.

(Users of [Caddy](#) are unaffected, since we believe it sets `X-Forwarded-Proto` by default.)

Upgrading to v1.27.0

Changes to callback URI for OAuth2 / OpenID Connect and SAML2

This version changes the URI used for callbacks from OAuth2 and SAML2 identity providers:

- If your server is configured for single sign-on via an OpenID Connect or OAuth2 identity provider, you will need to add `[synapse public baseurl]/_synapse/client/oidc/callback` to the list of permitted "redirect URLs" at the identity provider.

See the [OpenID docs](#) for more information on setting up OpenID Connect.

- If your server is configured for single sign-on via a SAML2 identity provider, you will need to add `[synapse public baseurl]/_synapse/client/saml2/authn_response` as a permitted "ACS location" (also known as "allowed callback URLs") at the identity provider.

The "Issuer" in the "AuthnRequest" to the SAML2 identity provider is also updated to `[synapse public baseurl]/_synapse/client/saml2/metadata.xml`. If your SAML2 identity provider uses this property to validate or otherwise identify Synapse, its configuration will need to be updated to use the new URL. Alternatively you could create a new, separate "EntityDescriptor" in your SAML2 identity provider with the new URLs and leave the URLs in the existing "EntityDescriptor" as they were.

Changes to HTML templates

The HTML templates for SSO and email notifications now have [Jinja2's autoescape](#) enabled for files ending in `.html`, `.htm`, and `.xml`. If you have customised these templates and see

issues when viewing them you might need to update them. It is expected that most configurations will need no changes.

If you have customised the templates *names* for these templates, it is recommended to verify they end in `.html` to ensure autoescape is enabled.

The above applies to the following templates:

- `add_threepid.html`
- `add_threepid_failure.html`
- `add_threepid_success.html`
- `notice_expiry.html`
- `notice_expiry.html`
- `notif_mail.html` (which, by default, includes `room.html` and `notif.html`)
- `password_reset.html`
- `password_reset_confirmation.html`
- `password_reset_failure.html`
- `password_reset_success.html`
- `registration.html`
- `registration_failure.html`
- `registration_success.html`
- `sso_account_deactivated.html`
- `sso_auth_bad_user.html`
- `sso_auth_confirm.html`
- `sso_auth_success.html`
- `sso_error.html`
- `sso_login_idp_picker.html`
- `sso_redirect_confirm.html`

Upgrading to v1.26.0

Rolling back to v1.25.0 after a failed upgrade

v1.26.0 includes a lot of large changes. If something problematic occurs, you may want to roll-back to a previous version of Synapse. Because v1.26.0 also includes a new database schema version, reverting that version is also required alongside the generic rollback instructions mentioned above. In short, to roll back to v1.25.0 you need to:

1. Stop the server
2. Decrease the schema version in the database:

```
UPDATE schema_version SET version = 58;
```

3. Delete the ignored users & chain cover data:

```
DROP TABLE IF EXISTS ignored_users;  
UPDATE rooms SET has_auth_chain_index = false;
```

For PostgreSQL run:

```
TRUNCATE event_auth_chain_links;  
TRUNCATE event_auth_chains;
```

For SQLite run:

```
DELETE FROM event_auth_chain_links;  
DELETE FROM event_auth_chains;
```

4. Mark the deltas as not run (so they will re-run on upgrade).

```
DELETE FROM applied_schema_deltas WHERE version = 59 AND file =  
"59/01ignored_user.py";  
DELETE FROM applied_schema_deltas WHERE version = 59 AND file =  
"59/06chain_cover_index.sql";
```

5. Downgrade Synapse by following the instructions for your installation method in the "Rolling back to older versions" section above.

Upgrading to v1.25.0

Last release supporting Python 3.5

This is the last release of Synapse which guarantees support with Python 3.5, which passed its upstream End of Life date several months ago.

We will attempt to maintain support through March 2021, but without guarantees.

In the future, Synapse will follow upstream schedules for ending support of older versions of Python and PostgreSQL. Please upgrade to at least Python 3.6 and PostgreSQL 9.6 as soon as possible.

Blacklisting IP ranges

Synapse v1.25.0 includes new settings, `ip_range_blacklist` and `ip_range_whitelist`, for controlling outgoing requests from Synapse for federation, identity servers, push, and for checking key validity for third-party invite events. The previous setting, `federation_ip_range_blacklist`, is deprecated. The new `ip_range_blacklist` defaults to private IP ranges if it is not defined.

If you have never customised `federation_ip_range_blacklist` it is recommended that you remove that setting.

If you have customised `federation_ip_range_blacklist` you should update the setting name to `ip_range_blacklist`.

If you have a custom push server that is reached via private IP space you may need to customise `ip_range_blacklist` or `ip_range_whitelist`.

Upgrading to v1.24.0

Custom OpenID Connect mapping provider breaking change

This release allows the OpenID Connect mapping provider to perform normalisation of the localpart of the Matrix ID. This allows for the mapping provider to specify different algorithms, instead of the [default way](#).

If your Synapse configuration uses a custom mapping provider (`oidc_config.user_mapping_provider.module` is specified and not equal to `synapse.handlers.oidc_handler.JinjaOidcMappingProvider`) then you *must* ensure that `map_user_attributes` of the mapping provider performs some normalisation of the `localpart` returned. To match previous behaviour you can use the `map_username_to_mxid_localpart` function provided by Synapse. An example is shown below:

```
from synapse.types import map_username_to_mxid_localpart

class MyMappingProvider:
    def map_user_attributes(self, userinfo, token):
        # ... your custom logic ...
        sso_user_id = ...
        localpart = map_username_to_mxid_localpart(sso_user_id)

        return {"localpart": localpart}
```

Removal historical Synapse Admin API

Historically, the Synapse Admin API has been accessible under:

- `/_matrix/client/api/v1/admin`
- `/_matrix/client/unstable/admin`
- `/_matrix/client/r0/admin`
- `/_synapse/admin/v1`

The endpoints with `/_matrix/client/*` prefixes have been removed as of v1.24.0. The Admin API is now only accessible under:

- `/_synapse/admin/v1`

The only exception is the `/admin/whois` endpoint, which is [also available via the client-server API](#).

The deprecation of the old endpoints was announced with Synapse 1.20.0 (released on 2020-09-22) and makes it easier for homeserver admins to lock down external access to the Admin API endpoints.

Upgrading to v1.23.0

Structured logging configuration breaking changes

This release deprecates use of the `structured: true` logging configuration for structured logging. If your logging configuration contains `structured: true` then it should be modified based on the [structured logging documentation](#).

The `structured` and `drains` logging options are now deprecated and should be replaced by standard logging configuration of `handlers` and `formatters`.

A future release of Synapse will make using `structured: true` an error.

Upgrading to v1.22.0

ThirdPartyEventRules breaking changes

This release introduces a backwards-incompatible change to modules making use of `ThirdPartyEventRules` in Synapse. If you make use of a module defined under the

`third_party_event_rules` config option, please make sure it is updated to handle the below change:

The `http_client` argument is no longer passed to modules as they are initialised. Instead, modules are expected to make use of the `http_client` property on the `ModuleApi` class. Modules are now passed a `module_api` argument during initialisation, which is an instance of `ModuleApi`. `ModuleApi` instances have a `http_client` property which acts the same as the `http_client` argument previously passed to `ThirdPartyEventRules` modules.

Upgrading to v1.21.0

Forwarding `/_synapse/client` through your reverse proxy

The [reverse proxy documentation](#) has been updated to include reverse proxy directives for `/_synapse/client/*` endpoints. As the user password reset flow now uses endpoints under this prefix, **you must update your reverse proxy configurations for user password reset to work.**

Additionally, note that the [Synapse worker documentation](#) has been updated to

: state that the `/_synapse/client/password_reset/email/submit_token` endpoint can be handled

by all workers. If you make use of Synapse's worker feature, please update your reverse proxy configuration to reflect this change.

New HTML templates

A new HTML template, `password_reset_confirmation.html`, has been added to the `synapse/res/templates` directory. If you are using a custom template directory, you may want to copy the template over and modify it.

Note that as of v1.20.0, templates do not need to be included in custom template directories for Synapse to start. The default templates will be used if a custom template cannot be found.

This page will appear to the user after clicking a password reset link that has been emailed to them.

To complete password reset, the page must include a way to make a `POST` request to `/_synapse/client/password_reset/{medium}/submit_token` with the query parameters from the original link, presented as a URL-encoded form. See the file itself for more details.

Updated Single Sign-on HTML Templates

The `saml_error.html` template was removed from Synapse and replaced with the `sso_error.html` template. If your Synapse is configured to use SAML and a custom `sso_redirect_confirm_template_dir` configuration then any customisations of the `saml_error.html` template will need to be merged into the `sso_error.html` template.

These templates are similar, but the parameters are slightly different:

- The `msg` parameter should be renamed to `error_description`.
- There is no longer a `code` parameter for the response code.
- A string `error` parameter is available that includes a short hint of why a user is seeing the error page.

Upgrading to v1.18.0

Docker `-py3` suffix will be removed in future versions

From 10th August 2020, we will no longer publish Docker images with the `-py3` tag suffix. The images tagged with the `-py3` suffix have been identical to the non-suffixed tags since release 0.99.0, and the suffix is obsolete.

On 10th August, we will remove the `latest-py3` tag. Existing per-release tags (such as `v1.18.0-py3`) will not be removed, but no new `-py3` tags will be added.

Scripts relying on the `-py3` suffix will need to be updated.

Redis replication is now recommended in lieu of TCP replication

When setting up worker processes, we now recommend the use of a Redis server for replication. **The old direct TCP connection method is deprecated and will be removed in a future release.** See the [worker documentation](#) for more details.

Upgrading to v1.14.0

This version includes a database update which is run as part of the upgrade, and which may take a couple of minutes in the case of a large server. Synapse will not respond to HTTP requests while this update is taking place.

Upgrading to v1.13.0

Incorrect database migration in old synapse versions

A bug was introduced in Synapse 1.4.0 which could cause the room directory to be incomplete or empty if Synapse was upgraded directly from v1.2.1 or earlier, to versions between v1.4.0 and v1.12.x.

This will *not* be a problem for Synapse installations which were:

: - created at v1.4.0 or later, - upgraded via v1.3.x, or - upgraded straight from v1.2.1 or earlier to v1.13.0 or later.

If completeness of the room directory is a concern, installations which are affected can be repaired as follows:

1. Run the following sql from a `psql` or `sqlite3` console:

```
INSERT INTO background_updates (update_name, progress_json, depends_on)
VALUES
  ('populate_stats_process_rooms', '{}',
  'current_state_events_membership');

INSERT INTO background_updates (update_name, progress_json, depends_on)
VALUES
  ('populate_stats_process_users', '{}', 'populate_stats_process_rooms');
```

2. Restart synapse.

New Single Sign-on HTML Templates

New templates (`sso_auth_confirm.html`, `sso_auth_success.html`, and `sso_account_deactivated.html`) were added to Synapse. If your Synapse is configured to

use SSO and a custom `sso_redirect_confirm_template_dir` configuration then these templates will need to be copied from `synapse/res/templates` into that directory.

Synapse SSO Plugins Method Deprecation

Plugins using the `complete_sso_login` method of `synapse.module_api.ModuleApi` should update to using the async/await version `complete_sso_login_async` which includes additional checks. The non-async version is considered deprecated.

Rolling back to v1.12.4 after a failed upgrade

v1.13.0 includes a lot of large changes. If something problematic occurs, you may want to roll-back to a previous version of Synapse. Because v1.13.0 also includes a new database schema version, reverting that version is also required alongside the generic rollback instructions mentioned above. In short, to roll back to v1.12.4 you need to:

1. Stop the server
2. Decrease the schema version in the database:

```
UPDATE schema_version SET version = 57;
```

3. Downgrade Synapse by following the instructions for your installation method in the "Rolling back to older versions" section above.

Upgrading to v1.12.0

This version includes a database update which is run as part of the upgrade, and which may take some time (several hours in the case of a large server). Synapse will not respond to HTTP requests while this update is taking place.

This is only likely to be a problem in the case of a server which is participating in many rooms.

0. As with all upgrades, it is recommended that you have a recent backup of your database which can be used for recovery in the event of any problems.
1. As an initial check to see if you will be affected, you can try running the following query from the `psql` or `sqlite3` console. It is safe to run it while Synapse is still running.

```

SELECT MAX(q.v) FROM (
  SELECT (
    SELECT ej.json AS v
    FROM state_events se INNER JOIN event_json ej USING (event_id)
    WHERE se.room_id=rooms.room_id AND se.type='m.room.create' AND
    se.state_key=''
    LIMIT 1
  ) FROM rooms WHERE rooms.room_version IS NULL
) q;

```

This query will take about the same amount of time as the upgrade process: ie, if it takes 5 minutes, then it is likely that Synapse will be unresponsive for 5 minutes during the upgrade.

If you consider an outage of this duration to be acceptable, no further action is necessary and you can simply start Synapse 1.12.0.

If you would prefer to reduce the downtime, continue with the steps below.

2. The easiest workaround for this issue is to manually create a new index before upgrading. On PostgreSQL, this can be done as follows:

```

CREATE INDEX CONCURRENTLY tmp_upgrade_1_12_0_index
ON state_events(room_id) WHERE type = 'm.room.create';

```

The above query may take some time, but is also safe to run while Synapse is running.

We assume that no SQLite users have databases large enough to be affected. If you *are* affected, you can run a similar query, omitting the `CONCURRENTLY` keyword. Note however that this operation may in itself cause Synapse to stop running for some time. Synapse admins are reminded that [SQLite is not recommended for use outside a test environment](#).

3. Once the index has been created, the `SELECT` query in step 1 above should complete quickly. It is therefore safe to upgrade to Synapse 1.12.0.
4. Once Synapse 1.12.0 has successfully started and is responding to HTTP requests, the temporary index can be removed:

```

DROP INDEX tmp_upgrade_1_12_0_index;

```

Upgrading to v1.10.0

Synapse will now log a warning on start up if used with a PostgreSQL database that has a non-recommended locale set.

See [Postgres](#) for details.

Upgrading to v1.8.0

Specifying a `log_file` config option will now cause Synapse to refuse to start, and should be replaced by with the `log_config` option. Support for the `log_file` option was removed in v1.3.0 and has since had no effect.

Upgrading to v1.7.0

In an attempt to configure Synapse in a privacy preserving way, the default behaviours of `allow_public_rooms_without_auth` and `allow_public_rooms_over_federation` have been inverted. This means that by default, only authenticated users querying the Client/Server API will be able to query the room directory, and relatedly that the server will not share room directory information with other servers over federation.

If your installation does not explicitly set these settings one way or the other and you want either setting to be `true` then it will necessary to update your homeserver configuration file accordingly.

For more details on the surrounding context see our [explainer](#).

Upgrading to v1.5.0

This release includes a database migration which may take several minutes to complete if there are a large number (more than a million or so) of entries in the `devices` table. This is only likely to a be a problem on very large installations.

Upgrading to v1.4.0

New custom templates

If you have configured a custom template directory with the `email.template_dir` option, be aware that there are new templates regarding registration and threepid management (see below) that must be included.

- `registration.html` and `registration.txt`
- `registration_success.html` and `registration_failure.html`
- `add_threepid.html` and `add_threepid.txt`
- `add_threepid_failure.html` and `add_threepid_success.html`

Synapse will expect these files to exist inside the configured template directory, and **will fail to start** if they are absent. To view the default templates, see [synapse/res/templates](#).

3pid verification changes

Note: As of this release, users will be unable to add phone numbers or email addresses to their accounts, without changes to the Synapse configuration. This includes adding an email address during registration.

It is possible for a user to associate an email address or phone number with their account, for a number of reasons:

- for use when logging in, as an alternative to the user id.
- in the case of email, as an alternative contact to help with account recovery.
- in the case of email, to receive notifications of missed messages.

Before an email address or phone number can be added to a user's account, or before such an address is used to carry out a password-reset, Synapse must confirm the operation with the owner of the email address or phone number. It does this by sending an email or text giving the user a link or token to confirm receipt. This process is known as '3pid verification'. ('3pid', or 'threepid', stands for third-party identifier, and we use it to refer to external identifiers such as email addresses and phone numbers.)

Previous versions of Synapse delegated the task of 3pid verification to an identity server by default. In most cases this server is [vector.im](#) or [matrix.org](#).

In Synapse 1.4.0, for security and privacy reasons, the homeserver will no longer delegate this task to an identity server by default. Instead, the server administrator will need to explicitly decide how they would like the verification messages to be sent.

In the medium term, the [vector.im](#) and [matrix.org](#) identity servers will disable support for delegated 3pid verification entirely. However, in order to ease the transition, they will retain the capability for a limited period. Delegated email verification will be disabled on Monday 2nd December 2019 (giving roughly 2 months notice). Disabling delegated SMS verification will follow some time after that once SMS verification support lands in Synapse.

Once delegated 3pid verification support has been disabled in the [vector.im](#) and [matrix.org](#) identity servers, all Synapse versions that depend on those instances will be unable to verify email and phone numbers through them. There are no imminent plans to remove delegated 3pid verification from Sydent generally. (Sydent is the identity server project that backs the [vector.im](#) and [matrix.org](#) instances).

Email

Following upgrade, to continue verifying email (e.g. as part of the registration process), admins can either:-

- Configure Synapse to use an email server.
- Run or choose an identity server which allows delegated email verification and delegate to it.

Configure SMTP in Synapse

To configure an SMTP server for Synapse, modify the configuration section headed `email`, and be sure to have at least the `smtp_host`, `smtp_port` and `notif_from` fields filled out.

You may also need to set `smtp_user`, `smtp_pass`, and `require_transport_security`.

See the [sample configuration file](#) for more details on these settings.

Delegate email to an identity server

Some admins will wish to continue using email verification as part of the registration process, but will not immediately have an appropriate SMTP server at hand.

To this end, we will continue to support email verification delegation via the `vector.im` and `matrix.org` identity servers for two months. Support for delegated email verification will be disabled on Monday 2nd December.

The `account_threepid_delegates` dictionary defines whether the homeserver should delegate an external server (typically an [identity server](#)) to handle sending confirmation messages via email and SMS.

So to delegate email verification, in `homeserver.yaml`, set `account_threepid_delegates.email` to the base URL of an identity server. For example:

```
account_threepid_delegates:  
  email: https://example.com      # Delegate email sending to example.com
```

Note that `account_threepid_delegates.email` replaces the deprecated `email.trust_identity_server_for_password_resets`: if `email.trust_identity_server_for_password_resets` is set to `true`, and `account_threepid_delegates.email` is not set, then the first entry in `trusted_third_party_id_servers` will be used as the `account_threepid_delegate` for email. This is to ensure compatibility with existing Synapse installs that set up external server handling for these tasks before v1.4.0. If `email.trust_identity_server_for_password_resets` is `true` and no trusted identity server domains are configured, Synapse will report an error and refuse to start.

If `email.trust_identity_server_for_password_resets` is `false` or absent and no `email` delegate is configured in `account_threepid_delegates`, then Synapse will send email verification messages itself, using the configured SMTP server (see above). that type.

Phone numbers

Synapse does not support phone-number verification itself, so the only way to maintain the ability for users to add phone numbers to their accounts will be by continuing to delegate phone number verification to the `matrix.org` and `vector.im` identity servers (or another identity server that supports SMS sending).

The `account_threepid_delegates` dictionary defines whether the homeserver should delegate an external server (typically an [identity server](#)) to handle sending confirmation messages via email and SMS.

So to delegate phone number verification, in `homeserver.yaml`, set `account_threepid_delegates.msisdn` to the base URL of an identity server. For example:

```
account_threepid_delegates:
  msisdn: https://example.com      # Delegate sms sending to example.com
```

The `matrix.org` and `vector.im` identity servers will continue to support delegated phone number verification via SMS until such time as it is possible for admins to configure their servers to perform phone number verification directly. More details will follow in a future release.

Rolling back to v1.3.1

If you encounter problems with v1.4.0, it should be possible to roll back to v1.3.1, subject to the following:

- The 'room statistics' engine was heavily reworked in this release (see [#5971](#)), including significant changes to the database schema, which are not easily reverted. This will cause the room statistics engine to stop updating when you downgrade.

The room statistics are essentially unused in v1.3.1 (in future versions of Synapse, they will be used to populate the room directory), so there should be no loss of functionality. However, the statistics engine will write errors to the logs, which can be avoided by setting the following in `homeserver.yaml`:

```
stats:
  enabled: false
```

Don't forget to re-enable it when you upgrade again, in preparation for its use in the room directory!

Upgrading to v1.2.0

Some counter metrics have been renamed, with the old names deprecated. See [the metrics documentation](#) for details.

Upgrading to v1.1.0

Synapse v1.1.0 removes support for older Python and PostgreSQL versions, as outlined in [our deprecation notice](#).

Minimum Python Version

Synapse v1.1.0 has a minimum Python requirement of Python 3.5. Python 3.6 or Python 3.7 are recommended as they have improved internal string handling, significantly reducing memory usage.

If you use current versions of the Matrix.org-distributed Debian packages or Docker images, action is not required.

If you install Synapse in a Python virtual environment, please see "Upgrading to v0.34.0" for notes on setting up a new virtualenv under Python 3.

Minimum PostgreSQL Version

If using PostgreSQL under Synapse, you will need to use PostgreSQL 9.5 or above. Please see the [PostgreSQL documentation](#) for more details on upgrading your database.

Upgrading to v1.0

Validation of TLS certificates

Synapse v1.0 is the first release to enforce validation of TLS certificates for the federation API. It is therefore essential that your certificates are correctly configured.

Note, v1.0 installations will also no longer be able to federate with servers that have not correctly configured their certificates.

In rare cases, it may be desirable to disable certificate checking: for example, it might be essential to be able to federate with a given legacy server in a closed federation. This can be done in one of two ways:-

- Configure the global switch `federation_verify_certificates` to `false`.
- Configure a whitelist of server domains to trust via `federation_certificate_verification_whitelist`.

See the [sample configuration file](#) for more details on these settings.

Email

When a user requests a password reset, Synapse will send an email to the user to confirm the request.

Previous versions of Synapse delegated the job of sending this email to an identity server. If the identity server was somehow malicious or became compromised, it would be theoretically possible to hijack an account through this means.

Therefore, by default, Synapse v1.0 will send the confirmation email itself. If Synapse is not configured with an SMTP server, password reset via email will be disabled.

To configure an SMTP server for Synapse, modify the configuration section headed `email`, and be sure to have at least the `smtp_host`, `smtp_port` and `notif_from` fields filled out. You may also need to set `smtp_user`, `smtp_pass`, and `require_transport_security`.

If you are absolutely certain that you wish to continue using an identity server for password resets, set `trust_identity_server_for_password_resets` to `true`.

See the [sample configuration file](#) for more details on these settings.

New email templates

Some new templates have been added to the default template directory for the purpose of the homeserver sending its own password reset emails. If you have configured a custom `template_dir` in your Synapse config, these files will need to be added.

`password_reset.html` and `password_reset.txt` are HTML and plain text templates respectively that contain the contents of what will be emailed to the user upon attempting to reset their password via email. `password_reset_success.html` and

`password_reset_failure.html` are HTML files that the content of which (assuming no redirect URL is set) will be shown to the user after they attempt to click the link in the email sent to them.

Upgrading to v0.99.0

Please be aware that, before Synapse v1.0 is released around March 2019, you will need to replace any self-signed certificates with those verified by a root CA. Information on how to do so can be found at the ACME docs.

Upgrading to v0.34.0

1. This release is the first to fully support Python 3. Synapse will now run on Python versions 3.5, or 3.6 (as well as 2.7). We recommend switching to Python 3, as it has been shown to give performance improvements.

For users who have installed Synapse into a virtualenv, we recommend doing this by creating a new virtualenv. For example:

```
virtualenv -p python3 ~/synapse/env3
source ~/synapse/env3/bin/activate
pip install matrix-synapse
```

You can then start synapse as normal, having activated the new virtualenv:

```
cd ~/synapse
source env3/bin/activate
synctl start
```

Users who have installed from distribution packages should see the relevant package documentation. See below for notes on Debian packages.

- o When upgrading to Python 3, you **must** make sure that your log files are configured as UTF-8, by adding `encoding: utf8` to the `RotatingFileHandler` configuration (if you have one) in your `<server>.log.config` file. For example, if your `log.config` file contains:

```

handlers:
  file:
    class: logging.handlers.RotatingFileHandler
    formatter: precise
    filename: homeserver.log
    maxBytes: 104857600
    backupCount: 10
    filters: [context]
  console:
    class: logging.StreamHandler
    formatter: precise
    filters: [context]

```

Then you should update this to be:

```

handlers:
  file:
    class: logging.handlers.RotatingFileHandler
    formatter: precise
    filename: homeserver.log
    maxBytes: 104857600
    backupCount: 10
    filters: [context]
    encoding: utf8
  console:
    class: logging.StreamHandler
    formatter: precise
    filters: [context]

```

There is no need to revert this change if downgrading to Python 2.

We are also making available Debian packages which will run Synapse on Python 3. You can switch to these packages with `apt-get install matrix-synapse-py3`, however, please read [debian/NEWS](#) before doing so. The existing `matrix-synapse` packages will continue to use Python 2 for the time being.

2. This release removes the `riot.im` from the default list of trusted identity servers.

If `riot.im` is in your homeserver's list of `trusted_third_party_id_servers`, you should remove it. It was added in case a hypothetical future identity server was put there. If you don't remove it, users may be unable to deactivate their accounts.

3. This release no longer installs the (unmaintained) Matrix Console web client as part of the default installation. It is possible to re-enable it by installing it separately and

setting the `web_client_location` config option, but please consider switching to another client.

Upgrading to v0.33.7

This release removes the example email notification templates from `res/templates` (they are now internal to the python package). This should only affect you if you (a) deploy your Synapse instance from a git checkout or a github snapshot URL, and (b) have email notifications enabled.

If you have email notifications enabled, you should ensure that `email.template_dir` is either configured to point at a directory where you have installed customised templates, or leave it unset to use the default templates.

Upgrading to v0.27.3

This release expands the anonymous usage stats sent if the opt-in `report_stats` configuration is set to `true`. We now capture RSS memory and cpu use at a very coarse level. This requires administrators to install the optional `psutil` python module.

We would appreciate it if you could assist by ensuring this module is available and `report_stats` is enabled. This will let us see if performance changes to synapse are having an impact to the general community.

Upgrading to v0.15.0

If you want to use the new URL previewing API (`/_matrix/media/r0/preview_url`) then you have to explicitly enable it in the config and update your dependencies. See README.rst for details.

Upgrading to v0.11.0

This release includes the option to send anonymous usage stats to matrix.org, and requires that administrators explicitly opt in or out by setting the `report_stats` option to either `true` or `false`.

We would really appreciate it if you could help our project out by reporting anonymized usage statistics from your homeserver. Only very basic aggregate data (e.g. number of users) will be reported, but it helps us to track the growth of the Matrix community, and helps us to make Matrix a success, as well as to convince other networks that they should peer with us.

Upgrading to v0.9.0

Application services have had a breaking API change in this version.

They can no longer register themselves with a home server using the AS HTTP API. This decision was made because a compromised application service with free reign to register any regex in effect grants full read/write access to the home server if a regex of `.*` is used. An attack where a compromised AS re-registers itself with `.*` was deemed too big of a security risk to ignore, and so the ability to register with the HS remotely has been removed.

It has been replaced by specifying a list of application service registrations in

`homeserver.yaml`:

```
app_service_config_files: ["registration-01.yaml", "registration-02.yaml"]
```

Where `registration-01.yaml` looks like:

```
url: <String> # e.g. "https://my.application.service.com"
as_token: <String>
hs_token: <String>
sender_localpart: <String> # This is a new field which denotes the user_id
localpart when using the AS token
namespaces:
  users:
    - exclusive: <Boolean>
      regex: <String> # e.g. "@prefix_.*"
  aliases:
    - exclusive: <Boolean>
      regex: <String>
  rooms:
    - exclusive: <Boolean>
      regex: <String>
```

Upgrading to v0.8.0

Servers which use captchas will need to add their public key to:

```
static/client/register/register_config.js

window.matrixRegistrationConfig = {
  recaptcha_public_key: "YOUR_PUBLIC_KEY"
};
```

This is required in order to support registration fallback (typically used on mobile devices).

Upgrading to v0.7.0

New dependencies are:

- pydenticon
- simplejson
- syutil
- matrix-angular-sdk

To pull in these dependencies in a virtual env, run:

```
python synapse/python_dependencies.py | xargs -n 1 pip install
```

Upgrading to v0.6.0

To pull in new dependencies, run:

```
python setup.py develop --user
```

This update includes a change to the database schema. To upgrade you first need to upgrade the database by running:

```
python scripts/upgrade_db_to_v0.6.0.py <db> <server_name> <signing_key>
```

Where `<db>` is the location of the database, `<server_name>` is the server name as specified in the synapse configuration, and `<signing_key>` is the location of the signing key as specified in the synapse configuration.

This may take some time to complete. Failures of signatures and content hashes can safely be ignored.

Upgrading to v0.5.1

Depending on precisely when you installed v0.5.0 you may have ended up with a stale release of the reference matrix webclient installed as a python module. To uninstall it and ensure you are depending on the latest module, please run:

```
$ pip uninstall syweb
```

Upgrading to v0.5.0

The webclient has been split out into a separate repository/package in this release. Before you restart your homeserver you will need to pull in the webclient package by running:

```
python setup.py develop --user
```

This release completely changes the database schema and so requires upgrading it before starting the new version of the homeserver.

The script "database-prepare-for-0.5.0.sh" should be used to upgrade the database. This will save all user information, such as logins and profiles, but will otherwise purge the database. This includes messages, which rooms the home server was a member of and room alias mappings.

If you would like to keep your history, please take a copy of your database file and ask for help in #matrix:matrix.org. The upgrade process is, unfortunately, non trivial and requires human intervention to resolve any resulting conflicts during the upgrade process.

Before running the command the homeserver should be first completely shutdown. To run it, simply specify the location of the database, e.g.:

```
./scripts/database-prepare-for-0.5.0.sh "homeserver.db"
```

Once this has successfully completed it will be safe to restart the homeserver. You may notice that the homeserver takes a few seconds longer to restart than usual as it reinitializes the database.

On startup of the new version, users can either rejoin remote rooms using room aliases or by being reinvited. Alternatively, if any other homeserver sends a message to a room that the homeserver was previously in the local HS will automatically rejoin the room.

Upgrading to v0.4.0

This release needs an updated syutil version. Run:

```
python setup.py develop
```

You will also need to upgrade your configuration as the signing key format has changed. Run:

```
python -m synapse.app.homeserver --config-path <CONFIG> --generate-config
```

Upgrading to v0.3.0

This registration API now closely matches the login API. This introduces a bit more backwards and forwards between the HS and the client, but this improves the overall

flexibility of the API. You can now GET on /register to retrieve a list of valid registration flows. Upon choosing one, they are submitted in the same way as login, e.g:

```
{
  type: m.login.password,
  user: foo,
  password: bar
}
```

The default HS supports 2 flows, with and without Identity Server email authentication. Enabling captcha on the HS will add in an extra step to all flows: `m.login.recaptcha` which must be completed before you can transition to the next stage. There is a new login type: `m.login.email.identity` which contains the `threepidCreds` key which were previously sent in the original register request. For more information on this, see the specification.

Web Client

The VoIP specification has changed between v0.2.0 and v0.3.0. Users should refresh any browser tabs to get the latest web client code. Users on v0.2.0 of the web client will not be able to call those on v0.3.0 and vice versa.

Upgrading to v0.2.0

The home server now requires setting up of SSL config before it can run. To automatically generate default config use:

```
$ python synapse/app/homeserver.py \
  --server-name machine.my.domain.name \
  --bind-port 8448 \
  --config-path homeserver.config \
  --generate-config
```

This config can be edited if desired, for example to specify a different SSL certificate to use. Once done you can run the home server using:

```
$ python synapse/app/homeserver.py --config-path homeserver.config
```

See the README.rst for more information.

Also note that some config options have been renamed, including:

- "host" to "server-name"
- "database" to "database-path"
- "port" to "bind-port" and "unsecure-port"

Upgrading to v0.0.1

This release completely changes the database schema and so requires upgrading it before starting the new version of the homeserver.

The script "database-prepare-for-0.0.1.sh" should be used to upgrade the database. This will save all user information, such as logins and profiles, but will otherwise purge the database. This includes messages, which rooms the home server was a member of and room alias mappings.

Before running the command the homeserver should be first completely shutdown. To run it, simply specify the location of the database, e.g.:

```
./scripts/database-prepare-for-0.0.1.sh "homeserver.db"
```

Once this has successfully completed it will be safe to restart the homeserver. You may notice that the homeserver takes a few seconds longer to restart than usual as it reinitializes the database.

On startup of the new version, users can either rejoin remote rooms using room aliases or by being reinvited. Alternatively, if any other homeserver sends a message to a room that the homeserver was previously in the local HS will automatically rejoin the room.

Setting up federation

Federation is the process by which users on different servers can participate in the same room. For this to work, those other servers must be able to contact yours to send messages.

The `server_name` configured in the Synapse configuration file (often `homeserver.yaml`) defines how resources (users, rooms, etc.) will be identified (eg: `@user:example.com`, `#room:example.com`). By default, it is also the domain that other servers will use to try to reach your server (via port 8448). This is easy to set up and will work provided you set the `server_name` to match your machine's public DNS hostname.

For this default configuration to work, you will need to listen for TLS connections on port 8448. The preferred way to do that is by using a reverse proxy: see [the reverse proxy documentation](#) for instructions on how to correctly set one up.

In some cases you might not want to run Synapse on the machine that has the `server_name` as its public DNS hostname, or you might want federation traffic to use a different port than 8448. For example, you might want to have your user names look like `@user:example.com`, but you want to run Synapse on `synapse.example.com` on port 443. This can be done using delegation, which allows an admin to control where federation traffic should be sent. See [the delegation documentation](#) for instructions on how to set this up.

Once federation has been configured, you should be able to join a room over federation. A good place to start is `#synapse:matrix.org` - a room for Synapse admins.

Troubleshooting

You can use the [federation tester](#) to check if your homeserver is configured correctly. Alternatively try the [JSON API used by the federation tester](#). Note that you'll have to modify this URL to replace `DOMAIN` with your `server_name`. Hitting the API directly provides extra detail.

The typical failure mode for federation is that when the server tries to join a room, it is rejected with "401: Unauthorized". Generally this means that other servers in the room could not access yours. (Joining a room over federation is a complicated dance which requires connections in both directions).

Another common problem is that people on other servers can't join rooms that you invite them to. This can be caused by an incorrectly-configured reverse proxy: see [the reverse proxy documentation](#) for instructions on how to correctly configure a reverse proxy.

Known issues

HTTP 308 Permanent Redirect redirects are not followed: Due to missing features in the HTTP library used by Synapse, 308 redirects are currently not followed by federating servers, which can cause `M_UNKNOWN` or `401 Unauthorized` errors. This may affect users who are redirecting apex-to-www (e.g. `example.com` -> `www.example.com`), and especially users of the Kubernetes *Nginx Ingress* module, which uses 308 redirect codes by default. For those Kubernetes users, [this Stackoverflow post](#) might be helpful. For other users, switching to a `301 Moved Permanently` code may be an option. 308 redirect codes will be supported properly in a future release of Synapse.

Running a demo federation of Synapses

If you want to get up and running quickly with a trio of homeservers in a private federation, there is a script in the `demo` directory. This is mainly useful just for development purposes. See [demo scripts](#).

Configuration

This section contains information on tweaking Synapse via the various options in the configuration file. A configuration file should have been generated when you [installed Synapse](#).

Configuring Synapse

This is intended as a guide to the Synapse configuration. The behavior of a Synapse instance can be modified through the many configuration settings documented here — each config option is explained, including what the default is, how to change the default and what sort of behaviour the setting governs. Also included is an example configuration for each setting. If you don't want to spend a lot of time thinking about options, the config as generated sets sensible defaults for all values. Do note however that the database defaults to SQLite, which is not recommended for production usage. You can read more on this subject [here](#).

Config Conventions

Configuration options that take a time period can be set using a number followed by a letter. Letters have the following meanings:

- **s** = second
- **m** = minute
- **h** = hour
- **d** = day
- **w** = week
- **y** = year

For example, setting `redaction_retention_period: 5m` would remove redacted messages from the database after 5 minutes, rather than 5 months.

In addition, configuration options referring to size use the following suffixes:

- **K** = KiB, or 1024 bytes
- **M** = MiB, or 1,048,576 bytes
- **G** = GiB, or 1,073,741,824 bytes
- **T** = TiB, or 1,099,511,627,776 bytes

For example, setting `max_avatar_size: 10M` means that Synapse will not accept files larger than 10,485,760 bytes for a user avatar.

Config Validation

The configuration file can be validated with the following command:

```
python -m synapse.config read <config key to print> -c <path to config>
```

To validate the entire file, omit `read <config key to print>`:

```
python -m synapse.config -c <path to config>
```

To see how to set other options, check the help reference:

```
python -m synapse.config --help
```

YAML

The configuration file is a [YAML](#) file, which means that certain syntax rules apply if you want your config file to be read properly. A few helpful things to know:

- `#` before any option in the config will comment out that setting and either a default (if available) will be applied or Synapse will ignore the setting. Thus, in example #1 below, the setting will be read and applied, but in example #2 the setting will not be read and a default will be applied.

Example #1:

```
pid_file: DATADIR/homeserver.pid
```

Example #2:

```
#pid_file: DATADIR/homeserver.pid
```

- Indentation matters! The indentation before a setting will determine whether a given setting is read as part of another setting, or considered on its own. Thus, in example #1, the `enabled` setting is read as a sub-option of the `presence` setting, and will be properly applied.

However, the lack of indentation before the `enabled` setting in example #2 means that when reading the config, Synapse will consider both `presence` and `enabled` as different settings. In this case, `presence` has no value, and thus a default applied, and `enabled` is an option that Synapse doesn't recognize and thus ignores.

Example #1:

```
presence:
  enabled: false
```

Example #2:

```
presence:
  enabled: false
```

In this manual, all top-level settings (ones with no indentation) are identified at the beginning of their section (i.e. "### `example_setting`") and the sub-options, if any, are identified and listed in the body of the section. In addition, each setting has an example of its usage, with the proper indentation shown.

Modules

Server admins can expand Synapse's functionality with external modules.

See [here](#) for more documentation on how to configure or create custom modules for Synapse.

modules

Use the `module` sub-option to add modules under this option to extend functionality. The `module` setting then has a sub-option, `config`, which can be used to define some configuration for the `module`.

Defaults to none.

Example configuration:

```
modules:
  - module: my_super_module.MySuperClass
    config:
      do_thing: true
  - module: my_other_super_module.SomeClass
    config: {}
```

Server

Define your homeserver name and other base options.

server_name

This sets the public-facing domain of the server.

The `server_name` name will appear at the end of usernames and room addresses created on your server. For example if the `server_name` was example.com, usernames on your

server would be in the format `@user:example.com`

In most cases you should avoid using a matrix specific subdomain such as `matrix.example.com` or `synapse.example.com` as the `server_name` for the same reasons you wouldn't use `user@email.example.com` as your email address. See [here](#) for information on how to host Synapse on a subdomain while preserving a clean `server_name`.

The `server_name` cannot be changed later so it is important to configure this correctly before you start Synapse. It should be all lowercase and may contain an explicit port.

There is no default for this option.

Example configuration #1:

```
server_name: matrix.org
```

Example configuration #2:

```
server_name: localhost:8080
```

pid_file

When running Synapse as a daemon, the file to store the pid in. Defaults to none.

Example configuration:

```
pid_file: DATADIR/homeserver.pid
```

daemonize

Specifies whether Synapse should be started as a daemon process. If Synapse is being managed by `systemd`, this option must be omitted or set to `false`.

This can also be set by the `--daemonize` (`-D`) argument when starting Synapse.

See `worker_daemonize` for more information on daemonizing workers.

Example configuration:

```
daemonize: true
```

print_pidfile

Print the path to the pidfile just before daemonizing. Defaults to false.

This can also be set by the `--print-pidfile` argument when starting Synapse.

Example configuration:

```
print_pidfile: true
```

user_agent_suffix

A suffix that is appended to the Synapse user-agent (ex. `Synapse/v1.123.0`). Defaults to None

Example configuration:

```
user_agent_suffix: " (I'm a teapot; Linux x86_64)"
```

use_frozen_dicts

Determines whether we should freeze the internal dict object in `FrozenEvent`. Freezing prevents bugs where we accidentally share e.g. signature dicts. However, freezing a dict is expensive. Defaults to false.

Example configuration:

```
use_frozen_dicts: true
```

web_client_location

The absolute URL to the web client which `/` will redirect to. Defaults to none.

Example configuration:

```
web_client_location: https://riot.example.com/
```

public_baseurl

The public-facing base URL that clients use to access this Homeserver (not including `_matrix/...`). This is the same URL a user might enter into the 'Custom Homeserver URL' field on their client. If you use Synapse with a reverse proxy, this should be the URL to reach Synapse via the proxy. Otherwise, it should be the URL to reach Synapse's client HTTP listener (see '[listeners](#)' below).

Defaults to `https://<server_name>/`.

Example configuration:

```
public_baseurl: https://example.com/
```

serve_server_wellknown

By default, other servers will try to reach our server on port 8448, which can be inconvenient in some environments.

Provided `https://<server_name>/` on port 443 is routed to Synapse, this option configures Synapse to serve a file at `https://<server_name>/.well-known/matrix/server`. This will tell other servers to send traffic to port 443 instead.

This option currently defaults to false.

See [Delegation of incoming federation traffic](#) for more information.

Example configuration:

```
serve_server_wellknown: true
```

extra_well_known_client_content

This option allows server runners to add arbitrary key-value pairs to the [client-facing `.well-known` response](#). Note that the `public_baseurl` config option must be provided for Synapse to serve a response to `/.well-known/matrix/client` at all.

If this option is provided, it parses the given yaml to json and serves it on `/.well-known/matrix/client` endpoint alongside the standard properties.

Added in Synapse 1.62.0.

Example configuration:

```
extra_well_known_client_content :  
  option1: value1  
  option2: value2
```

soft_file_limit

Set the soft limit on the number of file descriptors synapse can use. Zero is used to indicate synapse should set the soft limit to the hard limit. Defaults to 0.

Example configuration:

```
soft_file_limit: 3
```

presence

Presence tracking allows users to see the state (e.g online/offline) of other local and remote users. Set the `enabled` sub-option to false to disable presence tracking on this homeserver. Defaults to true. This option replaces the previous top-level 'use_presence' option.

Example configuration:

```
presence:  
  enabled: false  
  include_offline_users_on_sync: false
```

`enabled` can also be set to a special value of "untracked" which ignores updates received via clients and federation, while still accepting updates from the [module API](#).

The "untracked" option was added in Synapse 1.96.0.

When clients perform an initial or `full_state` sync, presence results for offline users are not included by default. Setting `include_offline_users_on_sync` to `true` will always include offline users in the results. Defaults to false.

require_auth_for_profile_requests

Whether to require authentication to retrieve profile data (avatars, display names) of other users through the client API. Defaults to false. Note that profile data is also available via the federation API, unless `allow_profile_lookup_over_federation` is set to false.

Example configuration:

```
require_auth_for_profile_requests: true
```

limit_profile_requests_to_users_who_share_rooms

Use this option to require a user to share a room with another user in order to retrieve their profile information. Only checked on Client-Server requests. Profile requests from other servers should be checked by the requesting server. Defaults to false.

Example configuration:

```
limit_profile_requests_to_users_who_share_rooms: true
```

include_profile_data_on_invite

Use this option to prevent a user's profile data from being retrieved and displayed in a room until they have joined it. By default, a user's profile data is included in an invite event, regardless of the values of the above two settings, and whether or not the users share a server. Defaults to true.

Example configuration:

```
include_profile_data_on_invite: false
```

allow_public_rooms_without_auth

If set to true, removes the need for authentication to access the server's public rooms directory through the client API, meaning that anyone can query the room directory. Defaults to false.

Example configuration:

```
allow_public_rooms_without_auth: true
```

allow_public_rooms_over_federation

If set to true, allows any other homeserver to fetch the server's public rooms directory via federation. Defaults to false.

Example configuration:

```
allow_public_rooms_over_federation: true
```

default_room_version

The default room version for newly created rooms on this server.

Known room versions are listed [here](#)

For example, for room version 1, `default_room_version` should be set to "1".

Currently defaults to "10".

Changed in Synapse 1.76: the default version room version was increased from 9 to 10.

Example configuration:

```
default_room_version: "8"
```

gc_thresholds

The garbage collection threshold parameters to pass to `gc.set_threshold`, if defined. Defaults to none.

Example configuration:

```
gc_thresholds: [700, 10, 10]
```

gc_min_interval

The minimum time in seconds between each GC for a generation, regardless of the GC thresholds. This ensures that we don't do GC too frequently. A value of `[1s, 10s, 30s]` indicates that a second must pass between consecutive generation 0 GCs, etc.

Defaults to `[1s, 10s, 30s]`.

Example configuration:

```
gc_min_interval: [0.5s, 30s, 1m]
```

filter_timeline_limit

Set the limit on the returned events in the timeline in the get and sync operations. Defaults to 100. A value of -1 means no upper limit.

Example configuration:

```
filter_timeline_limit: 5000
```

block_non_admin_invites

Whether room invites to users on this server should be blocked (except those sent by local server admins). Defaults to false.

Example configuration:

```
block_non_admin_invites: true
```

enable_search

If set to false, new messages will not be indexed for searching and users will receive errors when searching for messages. Defaults to true.

Example configuration:

```
enable_search: false
```

ip_range_blacklist

This option prevents outgoing requests from being sent to the specified blacklisted IP address CIDR ranges. If this option is not specified then it defaults to private IP address ranges (see the example below).

The blacklist applies to the outbound requests for federation, identity servers, push servers, and for checking key validity for third-party invite events.

(0.0.0.0 and :: are always blacklisted, whether or not they are explicitly listed here, since they correspond to unrouteable addresses.)

This option replaces `federation_ip_range_blacklist` in Synapse v1.25.0.

Note: The value is ignored when an HTTP proxy is in use.

Example configuration:

```
ip_range_blacklist:
- '127.0.0.0/8'
- '10.0.0.0/8'
- '172.16.0.0/12'
- '192.168.0.0/16'
- '100.64.0.0/10'
- '192.0.0.0/24'
- '169.254.0.0/16'
- '192.88.99.0/24'
- '198.18.0.0/15'
- '192.0.2.0/24'
- '198.51.100.0/24'
- '203.0.113.0/24'
- '224.0.0.0/4'
- '::1/128'
- 'fe80::/10'
- 'fc00::/7'
- '2001:db8::/32'
- 'ff00::/8'
- 'fec0::/10'
```

ip_range_whitelist

List of IP address CIDR ranges that should be allowed for federation, identity servers, push servers, and for checking key validity for third-party invite events. This is useful for specifying exceptions to wide-ranging blacklisted target IP ranges - e.g. for communication with a push server only visible in your network.

This whitelist overrides `ip_range_blacklist` and defaults to an empty list.

Example configuration:

```
ip_range_whitelist:
- '192.168.1.1'
```

listeners

List of ports that Synapse should listen on, their purpose and their configuration.

Sub-options for each listener include:

- `port`: the TCP port to bind to.
- `tag`: An alias for the port in the logger name. If set the tag is logged instead of the port. Default to `None`, is optional and only valid for listener with `type: http`. See the

docs [request log format](#).

- **bind_addresses**: a list of local addresses to listen on. The default is 'all local interfaces'.
- **type**: the type of listener. Normally `http`, but other valid options are:
 - `manhole`: (see the docs [here](#)),
 - `metrics`: (see the docs [here](#)),
- **tls**: set to true to enable TLS for this listener. Will use the TLS key/cert specified in `tls_private_key_path` / `tls_certificate_path`.
- **x_forwarded**: Only valid for an 'http' listener. Set to true to use the X-Forwarded-For header as the client IP. Useful when Synapse is behind a [reverse-proxy](#).
- **request_id_header**: The header extracted from each incoming request that is used as the basis for the request ID. The request ID is used in [logs](#) and tracing to correlate and match up requests. When unset, Synapse will automatically generate sequential request IDs. This option is useful when Synapse is behind a [reverse-proxy](#).

Added in Synapse 1.68.0.

- **resources**: Only valid for an 'http' listener. A list of resources to host on this port. Sub-options for each resource are:
 - `names`: a list of names of HTTP resources. See below for a list of valid resource names.
 - `compress`: set to true to enable gzip compression on HTTP bodies for this resource. This is currently only supported with the `client`, `consent`, `metrics` and `federation` resources.
- **additional_resources**: Only valid for an 'http' listener. A map of additional endpoints which should be loaded via dynamic modules.

Unix socket support (*Added in Synapse 1.89.0*):

- **path**: A path and filename for a Unix socket. Make sure it is located in a directory with read and write permissions, and that it already exists (the directory will not be created). Defaults to `None`.
 - **Note**: The use of both `path` and `port` options for the same `listener` is not compatible.
 - The `x_forwarded` option defaults to true when using Unix sockets and can be omitted.
 - Other options that would not make sense to use with a UNIX socket, such as `bind_addresses` and `tls` will be ignored and can be removed.

- **mode**: The file permissions to set on the UNIX socket. Defaults to `666`
- **Note**: Must be set as `type: http` (does not support `metrics` and `manhole`). Also make sure that `metrics` is not included in `resources -> names`

Valid resource names are:

- **client**: the client-server API (`/_matrix/client`). Also implies `media` and `static`. If configuring the main process, the Synapse Admin API (`/_synapse/admin`) is also implied.
- **consent**: user consent forms (`/_matrix/consent`). See [here](#) for more.
- **federation**: the server-server API (`/_matrix/federation`). Also implies `media`, `keys`, `openid`
- **keys**: the key discovery API (`/_matrix/key`).
- **media**: the media API (`/_matrix/media`).
- **metrics**: the metrics interface. See [here](#). (Not compatible with Unix sockets)
- **openid**: OpenID authentication. See [here](#).
- **replication**: the HTTP replication API (`/_synapse/replication`). See [here](#).
- **static**: static resources under `synapse/static` (`/_matrix/static`). (Mostly useful for 'fallback authentication').
- **health**: the [health check endpoint](#). This endpoint is by default active for all other resources and does not have to be activated separately. This is only useful if you want to use the health endpoint explicitly on a dedicated port or for `workers` and containers without listener e.g. [application services](#).

Example configuration #1:

```
listeners:
  # TLS-enabled listener: for when matrix traffic is sent directly to synapse.
  #
  # (Note that you will also need to give Synapse a TLS key and certificate: see
  the TLS section
  #
  - port: 8448
    type: http
    tls: true
    resources:
      - names: [client, federation]
```

Example configuration #2:

```

listeners:
  # Insecure HTTP listener: for when matrix traffic passes through a reverse
  # proxy
  # that unwraps TLS.
  #
  # If you plan to use a reverse proxy, please see
  # https://element-hq.github.io/synapse/latest/reverse_proxy.html.
  #
  - port: 8008
    tls: false
    type: http
    x_forwarded: true
    bind_addresses: ['::1', '127.0.0.1']

resources:
  - names: [client, federation]
    compress: false

  # example additional_resources:
  additional_resources:
    "/_matrix/my/custom/endpoint":
      module: my_module.CustomRequestHandler
      config: {}

  # Turn on the twisted ssh manhole service on localhost on the given
  # port.
  - port: 9000
    bind_addresses: ['::1', '127.0.0.1']
    type: manhole

```

Example configuration #3:

```

listeners:
  # Unix socket listener: Ideal for Synapse deployments behind a reverse proxy,
  # offering
  # lightweight interprocess communication without TCP/IP overhead, avoid port
  # conflicts, and providing enhanced security through system file permissions.
  #
  # Note that x_forwarded will default to true, when using a UNIX socket. Please
  # see
  # https://element-hq.github.io/synapse/latest/reverse_proxy.html.
  #
  - path: /run/synapse/main_public.sock
    type: http
    resources:
      - names: [client, federation]

```

manhole

Turn on the Twisted telnet manhole service on the given port. Defaults to none.

This can also be set by the `--manhole` argument when starting Synapse.

Example configuration:

```
manhole: 1234
```

manhole_settings

Connection settings for the manhole. You can find more information on the manhole [here](#). Manhole sub-options include:

- `username`: the username for the manhole. This defaults to 'matrix'.
- `password`: The password for the manhole. This defaults to 'rabbithole'.
- `ssh_priv_key_path` and `ssh_pub_key_path`: The private and public SSH key pair used to encrypt the manhole traffic. If these are left unset, then hardcoded and non-secret keys are used, which could allow traffic to be intercepted if sent over a public network.

Example configuration:

```
manhole_settings:  
  username: manhole  
  password: mypassword  
  ssh_priv_key_path: CONFDIR/id_rsa  
  ssh_pub_key_path: CONFDIR/id_rsa.pub
```

dummy_events_threshold

Forward extremities can build up in a room due to networking delays between homeservers. Once this happens in a large room, calculation of the state of that room can become quite expensive. To mitigate this, once the number of forward extremities reaches a given threshold, Synapse will send an `org.matrix.dummy_event` event, which will reduce the forward extremities in the room.

This setting defines the threshold (i.e. number of forward extremities in the room) at which dummy events are sent. The default value is 10.

Example configuration:

```
dummy_events_threshold: 5
```

delete_stale_devices_after

An optional duration. If set, Synapse will run a daily background task to log out and delete any device that hasn't been accessed for more than the specified amount of time.

Defaults to no duration, which means devices are never pruned.

Note: This task will always run on the main process, regardless of the value of `run_background_tasks_on`. This is due to workers currently not having the ability to delete devices.

Example configuration:

```
delete_stale_devices_after: 1y
```

email

Configuration for sending emails from Synapse.

Server admins can configure custom templates for email content. See [here](#) for more information.

This setting has the following sub-options:

- `smtp_host`: The hostname of the outgoing SMTP server to use. Defaults to 'localhost'.
- `smtp_port`: The port on the mail server for outgoing SMTP. Defaults to 465 if `force_tls` is true, else 25.

Changed in Synapse 1.64.0: the default port is now aware of `force_tls`.

- `smtp_user` and `smtp_pass`: Username/password for authentication to the SMTP server. By default, no authentication is attempted.
- `force_tls`: By default, Synapse connects over plain text and then optionally upgrades to TLS via STARTTLS. If this option is set to true, TLS is used from the start (Implicit TLS), and the option `require_transport_security` is ignored. It is recommended to enable this if supported by your mail server.

New in Synapse 1.64.0.

- `require_transport_security`: Set to true to require TLS transport security for SMTP. By default, Synapse will connect over plain text, and will then switch to TLS via STARTTLS *if the SMTP server supports it*. If this option is set, Synapse will refuse to connect unless the server supports STARTTLS.

- **enable_tls**: By default, if the server supports TLS, it will be used, and the server must present a certificate that is valid for `tlsname`. If this option is set to false, TLS will not be used.
- **tlsname**: The domain name the SMTP server's TLS certificate must be valid for, defaulting to `smtp_host`.
- **notif_from**: defines the "From" address to use when sending emails. It must be set if email sending is enabled. The placeholder '%(app)s' will be replaced by the application name, which is normally set in `app_name`, but may be overridden by the Matrix client application. Note that the placeholder must be written '%(app)s', including the trailing 's'.
- **app_name**: `app_name` defines the default value for '%(app)s' in `notif_from` and email subjects. It defaults to 'Matrix'.
- **enable_notifs**: Set to true to allow users to receive e-mail notifications. If this is not set, users can configure e-mail notifications but will not receive them. Disabled by default.
- **notif_for_new_users**: Set to false to disable automatic subscription to email notifications for new users. Enabled by default.
- **notif_delay_before_mail**: The time to wait before emailing about a notification. This gives the user a chance to view the message via push or an open client. Defaults to 10 minutes.

New in Synapse 1.99.0.

- **client_base_url**: Custom URL for client links within the email notifications. By default links will be based on "https://matrix.to". (This setting used to be called `riot_base_url`; the old name is still supported for backwards-compatibility but is now deprecated.)
- **validation_token_lifetime**: Configures the time that a validation email will expire after sending. Defaults to 1h.
- **invite_client_location**: The web client location to direct users to during an invite. This is passed to the identity server as the `org.matrix.web_client_location` key. Defaults to unset, giving no guidance to the identity server.
- **subjects**: Subjects to use when sending emails from Synapse. The placeholder '%(app)s' will be replaced with the value of the `app_name` setting, or by a value dictated by the Matrix client application. In addition, each subject can use the following placeholders: '%(person)s', which will be replaced by the displayname of the user(s) that sent the message(s), e.g. "Alice and Bob", and '%(room)s', which will be replaced by the name of the room the message(s) have been sent to, e.g. "My super room". In

addition, emails related to account administration will can use the '%(server_name)s' placeholder, which will be replaced by the value of the `server_name` setting in your Synapse configuration.

Here is a list of subjects for notification emails that can be set:

- `message_from_person_in_room`: Subject to use to notify about one message from one or more user(s) in a room which has a name. Defaults to "[%(app)s] You have a message on %(app)s from %(person)s in the %(room)s room..."
- `message_from_person`: Subject to use to notify about one message from one or more user(s) in a room which doesn't have a name. Defaults to "[%(app)s] You have a message on %(app)s from %(person)s..."
- `messages_from_person`: Subject to use to notify about multiple messages from one or more users in a room which doesn't have a name. Defaults to "[%(app)s] You have messages on %(app)s from %(person)s..."
- `messages_in_room`: Subject to use to notify about multiple messages in a room which has a name. Defaults to "[%(app)s] You have messages on %(app)s in the %(room)s room..."
- `messages_in_room_and_others`: Subject to use to notify about multiple messages in multiple rooms. Defaults to "[%(app)s] You have messages on %(app)s in the %(room)s room and others..."
- `messages_from_person_and_others`: Subject to use to notify about multiple messages from multiple persons in multiple rooms. This is similar to the setting above except it's used when the room in which the notification was triggered has no name. Defaults to "[%(app)s] You have messages on %(app)s from %(person)s and others..."
- `invite_from_person_to_room`: Subject to use to notify about an invite to a room which has a name. Defaults to "[%(app)s] %(person)s has invited you to join the %(room)s room on %(app)s..."
- `invite_from_person`: Subject to use to notify about an invite to a room which doesn't have a name. Defaults to "[%(app)s] %(person)s has invited you to chat on %(app)s..."
- `password_reset`: Subject to use when sending a password reset email. Defaults to "[%(server_name)s] Password reset"
- `email_validation`: Subject to use when sending a verification email to assert an address's ownership. Defaults to "[%(server_name)s] Validate your email"

Example configuration:

```

email:
  smtp_host: mail.server
  smtp_port: 587
  smtp_user: "exampleusername"
  smtp_pass: "examplepassword"
  force_tls: true
  require_transport_security: true
  enable_tls: false
  tlsname: mail.server.example.com
  notif_from: "Your Friendly %(app)s homeserver <noreply@example.com>"
  app_name: my_branded_matrix_server
  enable_notifs: true
  notif_for_new_users: false
  client_base_url: "http://localhost/riot"
  validation_token_lifetime: 15m
  invite_client_location: https://app.element.io

subjects:
  message_from_person_in_room: "[%(app)s] You have a message on %(app)s from %(person)s in the %(room)s room..."
  message_from_person: "[%(app)s] You have a message on %(app)s from %(person)s..."
  messages_from_person: "[%(app)s] You have messages on %(app)s from %(person)s..."
  messages_in_room: "[%(app)s] You have messages on %(app)s in the %(room)s room..."
  messages_in_room_and_others: "[%(app)s] You have messages on %(app)s in the %(room)s room and others..."
  messages_from_person_and_others: "[%(app)s] You have messages on %(app)s from %(person)s and others..."
  invite_from_person_to_room: "[%(app)s] %(person)s has invited you to join the %(room)s room on %(app)s..."
  invite_from_person: "[%(app)s] %(person)s has invited you to chat on %(app)s..."
  password_reset: "[%(server_name)s] Password reset"
  email_validation: "[%(server_name)s] Validate your email"

```

max_event_delay_duration

The maximum allowed duration by which sent events can be delayed, as per [MSC4140](#). Must be a positive value if set.

Defaults to no duration (`null`), which disallows sending delayed events.

Example configuration:

```
max_event_delay_duration: 24h
```

Homeserver blocking

Useful options for Synapse admins.

admin_contact

How to reach the server admin, used in `ResourceLimitError`. Defaults to none.

Example configuration:

```
admin_contact: 'mailto:admin@server.com'
```

hs_disabled and hs_disabled_message

Blocks users from connecting to the homeserver and provides a human-readable reason why the connection was blocked. Defaults to false.

Example configuration:

```
hs_disabled: true
hs_disabled_message: 'Reason for why the HS is blocked'
```

limit_usage_by_mau

This option disables/enables monthly active user blocking. Used in cases where the admin or server owner wants to limit the number of monthly active users. When enabled and a limit is reached the server returns a `ResourceLimitError` with error type `Codes.RESOURCE_LIMIT_EXCEEDED`. Defaults to false. If this is enabled, a value for `max_mau_value` must also be set.

See [Monthly Active Users](#) for details on how to configure MAU.

Example configuration:

```
limit_usage_by_mau: true
```

max_mau_value

This option sets the hard limit of monthly active users above which the server will start blocking user actions if `limit_usage_by_mau` is enabled. Defaults to 0.

Example configuration:

```
max_mau_value: 50
```

mau_trial_days

The option `mau_trial_days` is a means to add a grace period for active users. It means that users must be active for the specified number of days before they can be considered active and guards against the case where lots of users sign up in a short space of time never to return after their initial session. Defaults to 0.

Example configuration:

```
mau_trial_days: 5
```

mau_appservice_trial_days

The option `mau_appservice_trial_days` is similar to `mau_trial_days`, but applies a different trial number if the user was registered by an appservice. A value of 0 means no trial days are applied. Appservices not listed in this dictionary use the value of `mau_trial_days` instead.

Example configuration:

```
mau_appservice_trial_days:  
  my_appservice_id: 3  
  another_appservice_id: 6
```

mau_limit_alerting

The option `mau_limit_alerting` is a means of limiting client-side alerting should the mau limit be reached. This is useful for small instances where the admin has 5 mau seats (say) for 5 specific people and no interest increasing the mau limit further. Defaults to true, which means that alerting is enabled.

Example configuration:

```
mau_limit_alerting: false
```

mau_stats_only

If enabled, the metrics for the number of monthly active users will be populated, however no one will be limited based on these numbers. If `limit_usage_by_mau` is true, this is implied to be true. Defaults to false.

Example configuration:

```
mau_stats_only: true
```

mau_limit_reserved_threepids

Sometimes the server admin will want to ensure certain accounts are never blocked by mau checking. These accounts are specified by this option. Defaults to none. Add accounts by specifying the `medium` and `address` of the reserved threepid (3rd party identifier).

Example configuration:

```
mau_limit_reserved_threepids:
  - medium: 'email'
    address: 'reserved_user@example.com'
```

server_context

This option is used by phonehome stats to group together related servers. Defaults to none.

Example configuration:

```
server_context: context
```

limit_remote_rooms

When this option is enabled, the room "complexity" will be checked before a user joins a new remote room. If it is above the complexity limit, the server will disallow joining, or will instantly leave. This is useful for homeservers that are resource-constrained. Options for this setting include:

- `enabled`: whether this check is enabled. Defaults to false.
- `complexity`: the limit above which rooms cannot be joined. The default is 1.0.
- `complexity_error`: override the error which is returned when the room is too complex with a custom message.
- `admins_can_join`: allow server admins to join complex rooms. Default is false.

Room complexity is an arbitrary measure based on factors such as the number of users in the room.

Example configuration:

```
limit_remote_rooms:
  enabled: true
  complexity: 0.5
  complexity_error: "I can't let you do that, Dave."
  admins_can_join: true
```

require_membership_for_aliases

Whether to require a user to be in the room to add an alias to it. Defaults to true.

Example configuration:

```
require_membership_for_aliases: false
```

allow_per_room_profiles

Whether to allow per-room membership profiles through the sending of membership events with profile information that differs from the target's global profile. Defaults to true.

Example configuration:

```
allow_per_room_profiles: false
```

max_avatar_size

The largest permissible file size in bytes for a user avatar. Defaults to no restriction. Use M for MB and K for KB.

Note that user avatar changes will not work if this is set without using Synapse's media repository.

Example configuration:

```
max_avatar_size: 10M
```

allowed_avatar_mimetypes

The MIME types allowed for user avatars. Defaults to no restriction.

Note that user avatar changes will not work if this is set without using Synapse's media repository.

Example configuration:

```
allowed_avatar_mimetypes: ["image/png", "image/jpeg", "image/gif"]
```

redaction_retention_period

How long to keep redacted events in unredacted form in the database. After this period redacted events get replaced with their redacted form in the DB.

Synapse will check whether the retention period has concluded for redacted events every 5 minutes. Thus, even if this option is set to `0`, Synapse may still take up to 5 minutes to purge redacted events from the database.

Defaults to `7d`. Set to `null` to disable.

Example configuration:

```
redaction_retention_period: 28d
```

forgotten_room_retention_period

How long to keep locally forgotten rooms before purging them from the DB.

Defaults to `null`, meaning it's disabled.

Example configuration:

```
forgotten_room_retention_period: 28d
```

user_ips_max_age

How long to track users' last seen time and IPs in the database.

Defaults to `28d`. Set to `null` to disable clearing out of old rows.

Example configuration:

```
user_ips_max_age: 14d
```

request_token_inhibit_3pid_errors

Inhibits the `/requestToken` endpoints from returning an error that might leak information about whether an e-mail address is in use or not on this homeserver. Defaults to false. Note that for some endpoints the error situation is the e-mail already being used, and for others the error is entering the e-mail being unused. If this option is enabled, instead of returning an error, these endpoints will act as if no error happened and return a fake session ID ('sid') to clients.

Example configuration:

```
request_token_inhibit_3pid_errors: true
```

next_link_domain_whitelist

A list of domains that the domain portion of `next_link` parameters must match.

This parameter is optionally provided by clients while requesting validation of an email or phone number, and maps to a link that users will be automatically redirected to after validation succeeds. Clients can make use this parameter to aid the validation process.

The whitelist is applied whether the homeserver or an identity server is handling validation.

The default value is no whitelist functionality; all domains are allowed. Setting this value to an empty list will instead disallow all domains.

Example configuration:

```
next_link_domain_whitelist: ["matrix.org"]
```

templates and custom_template_directory

These options define templates to use when generating email or HTML page contents. The `custom_template_directory` determines which directory Synapse will try to find template files in to use to generate email or HTML page contents. If not set, or a file is not found within the template directory, a default template from within the Synapse package will be used.

See [here](#) for more information about using custom templates.

Example configuration:

```
templates:  
  custom_template_directory: /path/to/custom/templates/
```

retention

This option and the associated options determine message retention policy at the server level.

Room admins and mods can define a retention period for their rooms using the `m.room.retention` state event, and server admins can cap this period by setting the `allowed_lifetime_min` and `allowed_lifetime_max` config options.

If this feature is enabled, Synapse will regularly look for and purge events which are older than the room's maximum retention period. Synapse will also filter events received over federation so that events that should have been purged are ignored and not stored again.

The message retention policies feature is disabled by default. You can read more about this feature [here](#).

This setting has the following sub-options:

- `default_policy`: Default retention policy. If set, Synapse will apply it to rooms that lack the 'm.room.retention' state event. This option is further specified by the `min_lifetime` and `max_lifetime` sub-options associated with it. Note that the value of `min_lifetime` doesn't matter much because Synapse doesn't take it into account yet.
- `allowed_lifetime_min` and `allowed_lifetime_max`: Retention policy limits. If set, and the state of a room contains a `m.room.retention` event in its state which contains a `min_lifetime` or a `max_lifetime` that's out of these bounds, Synapse will cap the room's policy to these limits when running purge jobs.
- `purge_jobs` and the associated `shortest_max_lifetime` and `longest_max_lifetime` sub-options: Server admins can define the settings of the background jobs purging the

events whose lifetime has expired under the `purge_jobs` section.

If no configuration is provided for this option, a single job will be set up to delete expired events in every room daily.

Each job's configuration defines which range of message lifetimes the job takes care of. For example, if `shortest_max_lifetime` is '2d' and `longest_max_lifetime` is '3d', the job will handle purging expired events in rooms whose state defines a `max_lifetime` that's both higher than 2 days, and lower than or equal to 3 days. Both the minimum and the maximum value of a range are optional, e.g. a job with no `shortest_max_lifetime` and a `longest_max_lifetime` of '3d' will handle every room with a retention policy whose `max_lifetime` is lower than or equal to three days.

The rationale for this per-job configuration is that some rooms might have a retention policy with a low `max_lifetime`, where history needs to be purged of outdated messages on a more frequent basis than for the rest of the rooms (e.g. every 12h), but not want that purge to be performed by a job that's iterating over every room it knows, which could be heavy on the server.

If any purge job is configured, it is strongly recommended to have at least a single job with neither `shortest_max_lifetime` nor `longest_max_lifetime` set, or one job without `shortest_max_lifetime` and one job without `longest_max_lifetime` set. Otherwise some rooms might be ignored, even if `allowed_lifetime_min` and `allowed_lifetime_max` are set, because capping a room's policy to these values is done after the policies are retrieved from Synapse's database (which is done using the range specified in a purge job's configuration).

Example configuration:

```
retention:
  enabled: true
  default_policy:
    min_lifetime: 1d
    max_lifetime: 1y
  allowed_lifetime_min: 1d
  allowed_lifetime_max: 1y
purge_jobs:
  - longest_max_lifetime: 3d
    interval: 12h
  - shortest_max_lifetime: 3d
    interval: 1d
```

TLS

Options related to TLS.

tls_certificate_path

This option specifies a PEM-encoded X509 certificate for TLS. This certificate, as of Synapse 1.0, will need to be a valid and verifiable certificate, signed by a recognised Certificate Authority. Defaults to none.

Be sure to use a `.pem` file that includes the full certificate chain including any intermediate certificates (for instance, if using certbot, use `fullchain.pem` as your certificate, not `cert.pem`).

Example configuration:

```
tls_certificate_path: "CONFDIR/SERVERNAME.tls.crt"
```

tls_private_key_path

PEM-encoded private key for TLS. Defaults to none.

Example configuration:

```
tls_private_key_path: "CONFDIR/SERVERNAME.tls.key"
```

federation_verify_certificates

Whether to verify TLS server certificates for outbound federation requests.

Defaults to true. To disable certificate verification, set the option to false.

Example configuration:

```
federation_verify_certificates: false
```

federation_client_minimum_tls_version

The minimum TLS version that will be used for outbound federation requests.

Defaults to `"1"`. Configurable to `"1"`, `"1.1"`, `"1.2"`, or `"1.3"`. Note that setting this value higher than `"1.2"` will prevent federation to most of the public Matrix network: only configure it to `"1.3"` if you have an entirely private federation setup and you can ensure TLS 1.3 support.

Example configuration:

```
federation_client_minimum_tls_version: "1.2"
```

federation_certificate_verification_whitelist

Skip federation certificate verification on a given whitelist of domains.

This setting should only be used in very specific cases, such as federation over Tor hidden services and similar. For private networks of homeservers, you likely want to use a private CA instead.

Only effective if `federation_verify_certificates` is `true`.

Example configuration:

```
federation_certificate_verification_whitelist:
- lon.example.com
- "*.domain.com"
- "*.onion"
```

federation_custom_ca_list

List of custom certificate authorities for federation traffic.

This setting should only normally be used within a private network of homeservers.

Note that this list will replace those that are provided by your operating environment. Certificates must be in PEM format.

Example configuration:

```
federation_custom_ca_list:
- myCA1.pem
- myCA2.pem
- myCA3.pem
```

Federation

Options related to federation.

federation_domain_whitelist

Restrict federation to the given whitelist of domains. N.B. we recommend also firewalling your federation listener to limit inbound federation traffic as early as possible, rather than relying purely on this application-layer restriction. If not specified, the default is to whitelist everything.

Note: this does not stop a server from joining rooms that servers not on the whitelist are in. As such, this option is really only useful to establish a "private federation", where a group of servers all whitelist each other and have the same whitelist.

Example configuration:

```
federation_domain_whitelist:
  - lon.example.com
  - nyc.example.com
  - syd.example.com
```

federation_whitelist_endpoint_enabled

Enables an endpoint for fetching the federation whitelist config.

The request method and path is `GET /_synapse/client/v1/config/federation_whitelist`, and the response format is:

```
{
  "whitelist_enabled": true, // Whether the federation whitelist is being enforced
  "whitelist": [ // Which server names are allowed by the whitelist
    "example.com"
  ]
}
```

If `whitelist_enabled` is `false` then the server is permitted to federate with all others.

The endpoint requires authentication.

Example configuration:

```
federation_whitelist_endpoint_enabled: true
```

federation_metrics_domains

Report prometheus metrics on the age of PDUs being sent to and received from the given domains. This can be used to give an idea of "delay" on inbound and outbound federation,

though be aware that any delay can be due to problems at either end or with the intermediate network.

By default, no domains are monitored in this way.

Example configuration:

```
federation_metrics_domains:
  - matrix.org
  - example.com
```

allow_profile_lookup_over_federation

Set to false to disable profile lookup over federation. By default, the Federation API allows other homeservers to obtain profile data of any user on this homeserver.

Example configuration:

```
allow_profile_lookup_over_federation: false
```

allow_device_name_lookup_over_federation

Set this option to true to allow device display name lookup over federation. By default, the Federation API prevents other homeservers from obtaining the display names of any user devices on this homeserver.

Example configuration:

```
allow_device_name_lookup_over_federation: true
```

federation

The federation section defines some sub-options related to federation.

The following options are related to configuring timeout and retry logic for one request, independently of the others. Short retry algorithm is used when something or someone will wait for the request to have an answer, while long retry is used for requests that happen in the background, like sending a federation transaction.

- `client_timeout`: timeout for the federation requests. Default to 60s.
- `max_short_retry_delay`: maximum delay to be used for the short retry algo. Default to 2s.

- `max_long_retry_delay`: maximum delay to be used for the short retry algo. Default to 60s.
- `max_short_retries`: maximum number of retries for the short retry algo. Default to 3 attempts.
- `max_long_retries`: maximum number of retries for the long retry algo. Default to 10 attempts.

The following options control the retry logic when communicating with a specific homeserver destination. Unlike the previous configuration options, these values apply across all requests for a given destination and the state of the backoff is stored in the database.

- `destination_min_retry_interval`: the initial backoff, after the first request fails. Defaults to 10m.
- `destination_retry_multiplier`: how much we multiply the backoff by after each subsequent fail. Defaults to 2.
- `destination_max_retry_interval`: a cap on the backoff. Defaults to a week.

Example configuration:

```
federation:
  client_timeout: 180s
  max_short_retry_delay: 7s
  max_long_retry_delay: 100s
  max_short_retries: 5
  max_long_retries: 20
  destination_min_retry_interval: 30s
  destination_retry_multiplier: 5
  destination_max_retry_interval: 12h
```

Caching

Options related to caching.

`event_cache_size`

The number of events to cache in memory. Defaults to 10K. Like other caches, this is affected by `caches.global_factor` (see below).

For example, the default is 10K and the `global_factor` default is 0.5.

Since $10K * 0.5$ is 5K then the event cache size will be 5K.

The cache affected by this configuration is named as "`getEvent`".

Note that this option is not part of the `caches` section.

Example configuration:

```
event_cache_size: 15K
```

caches and associated values

A cache 'factor' is a multiplier that can be applied to each of Synapse's caches in order to increase or decrease the maximum number of entries that can be stored.

`caches` can be configured through the following sub-options:

- `global_factor`: Controls the global cache factor, which is the default cache factor for all caches if a specific factor for that cache is not otherwise set.

This can also be set by the `SYNAPSE_CACHE_FACTOR` environment variable. Setting by environment variable takes priority over setting through the config file.

Defaults to 0.5, which will halve the size of all caches.

Note that changing this value also affects the HTTP connection pool.

- `per_cache_factors`: A dictionary of cache name to cache factor for that individual cache. Overrides the global cache factor for a given cache.

These can also be set through environment variables comprised of `SYNAPSE_CACHE_FACTOR_` + the name of the cache in capital letters and underscores. Setting by environment variable takes priority over setting through the config file. Ex. `SYNAPSE_CACHE_FACTOR_GET_USERS_WHO_SHARE_ROOM_WITH_USER=2.0`

Some caches have '*' and other characters that are not alphanumeric or underscores. These caches can be named with or without the special characters stripped. For example, to specify the cache factor for `*stateGroupCache*` via an environment variable would be `SYNAPSE_CACHE_FACTOR_STATEGROUPCACHE=2.0`.

- `expire_caches`: Controls whether cache entries are evicted after a specified time period. Defaults to true. Set to false to disable this feature. Note that never expiring caches may result in excessive memory usage.
- `cache_entry_ttl`: If `expire_caches` is enabled, this flag controls how long an entry can be in a cache without having been accessed before being evicted. Defaults to 30m.
- `sync_response_cache_duration`: Controls how long the results of a /sync request are cached for after a successful response is returned. A higher duration can help clients

with intermittent connections, at the cost of higher memory usage. A value of zero means that sync responses are not cached. Defaults to 2m.

Changed in Synapse 1.62.0: The default was changed from 0 to 2m.

- `cache_autotuning` and its sub-options `max_cache_memory_usage`, `target_cache_memory_usage`, and `min_cache_ttl` work in conjunction with each other to maintain a balance between cache memory usage and cache entry availability. You must be using `jemalloc` to utilize this option, and all three of the options must be specified for this feature to work. This option defaults to off, enable it by providing values for the sub-options listed below. Please note that the feature will not work and may cause unstable behavior (such as excessive emptying of caches or exceptions) if all of the values are not provided. Please see the [Config Conventions](#) for information on how to specify memory size and cache expiry durations.
 - `max_cache_memory_usage` sets a ceiling on how much memory the cache can use before caches begin to be continuously evicted. They will continue to be evicted until the memory usage drops below the `target_cache_memory_usage`, set in the setting below, or until the `min_cache_ttl` is hit. There is no default value for this option.
 - `target_cache_memory_usage` sets a rough target for the desired memory usage of the caches. There is no default value for this option.
 - `min_cache_ttl` sets a limit under which newer cache entries are not evicted and is only applied when caches are actively being evicted/`max_cache_memory_usage` has been exceeded. This is to protect hot caches from being emptied while Synapse is evicting due to memory. There is no default value for this option.

Example configuration:

```
event_cache_size: 15K
caches:
  global_factor: 1.0
  per_cache_factors:
    get_users_who_share_room_with_user: 2.0
  sync_response_cache_duration: 2m
  cache_autotuning:
    max_cache_memory_usage: 1024M
    target_cache_memory_usage: 758M
    min_cache_ttl: 5m
```

Reloading cache factors

The cache factors (i.e. `caches.global_factor` and `caches.per_cache_factors`) may be reloaded at any time by sending a `SIGHUP` signal to Synapse using e.g.

```
kill -HUP [PID_OF_SYNAPSE_PROCESS]
```

If you are running multiple workers, you must individually update the worker config file and send this signal to each worker process.

If you're using the [example systemd service](#) file in Synapse's `contrib` directory, you can send a `SIGHUP` signal by using `systemctl reload matrix-synapse`.

Database

Config options related to database settings.

database

The `database` setting defines the database that synapse uses to store all of its data.

Associated sub-options:

- `name`: this option specifies the database engine to use: either `sqlite3` (for SQLite) or `psycopg2` (for PostgreSQL). If no name is specified Synapse will default to SQLite.
- `txn_limit` gives the maximum number of transactions to run per connection before reconnecting. Defaults to 0, which means no limit.
- `allow_unsafe_locale` is an option specific to Postgres. Under the default behavior, Synapse will refuse to start if the postgres db is set to a non-C locale. You can override this behavior (which is *not* recommended) by setting `allow_unsafe_locale` to true. Note that doing so may corrupt your database. You can find more information [here](#) and [here](#).
- `args` gives options which are passed through to the database engine, except for options starting with `cp_`, which are used to configure the Twisted connection pool. For a reference to valid arguments, see:
 - for [sqlite](#)
 - for [postgres](#)
 - for [the connection pool](#)

For more information on using Synapse with Postgres, see [here](#).

Example SQLite configuration:

```
database:
  name: sqlite3
  args:
    database: /path/to/homeserver.db
```

Example Postgres configuration:

```
database:
  name: psycopg2
  txn_limit: 10000
  args:
    user: synapse_user
    password: secretpassword
    dbname: synapse
    host: localhost
    port: 5432
    cp_min: 5
    cp_max: 10
```

databases

The `databases` option allows specifying a mapping between certain database tables and database host details, spreading the load of a single Synapse instance across multiple database backends. This is often referred to as "database sharding". This option is only supported for PostgreSQL database backends.

Important note: This is a supported option, but is not currently used in production by the Matrix.org Foundation. Proceed with caution and always make backups.

`databases` is a dictionary of arbitrarily-named database entries. Each entry is equivalent to the value of the `database` homeserver config option (see above), with the addition of a `data_stores` key. `data_stores` is an array of strings that specifies the data store(s) (a defined label for a set of tables) that should be stored on the associated database backend entry.

The currently defined values for `data_stores` are:

- `"state"`: Database that relates to state groups will be stored in this database.

Specifically, that means the following tables:

- `state_groups`
- `state_group_edges`
- `state_groups_state`

And the following sequences:

- `state_groups_seq_id`

- **"main"**: All other database tables and sequences.

All databases will end up with additional tables used for tracking database schema migrations and any pending background updates. Synapse will create these automatically on startup when checking for and/or performing database schema migrations.

To migrate an existing database configuration (e.g. all tables on a single database) to a different configuration (e.g. the "main" data store on one database, and "state" on another), do the following:

1. Take a backup of your existing database. Things can and do go wrong and database corruption is no joke!
2. Ensure all pending database migrations have been applied and background updates have run. The simplest way to do this is to use the `update_synapse_database` script supplied with your Synapse installation.

```
update_synapse_database --database-config homeserver.yaml --run-background-updates
```

3. Copy over the necessary tables and sequences from one database to the other. Tables relating to database migrations, schemas, schema versions and background updates should **not** be copied.

As an example, say that you'd like to split out the "state" data store from an existing database which currently contains all data stores.

Simply copy the tables and sequences defined above for the "state" datastore from the existing database to the secondary database. As noted above, additional tables will be created in the secondary database when Synapse is started.

4. Modify/create the `databases` option in your `homeserver.yaml` to match the desired database configuration.
5. Start Synapse. Check that it starts up successfully and that things generally seem to be working.
6. Drop the old tables that were copied in step 3.

Only one of the options `database` or `databases` may be specified in your config, but not both.

Example configuration:

```

databases:
  basement_box:
    name: psycopg2
    txn_limit: 10000
    data_stores: ["main"]
  args:
    user: synapse_user
    password: secretpassword
    dbname: synapse_main
    host: localhost
    port: 5432
    cp_min: 5
    cp_max: 10

my_other_database:
  name: psycopg2
  txn_limit: 10000
  data_stores: ["state"]
  args:
    user: synapse_user
    password: secretpassword
    dbname: synapse_state
    host: localhost
    port: 5432
    cp_min: 5
    cp_max: 10

```

Logging

Config options related to logging.

log_config

This option specifies a yaml python logging config file as described [here](#).

Example configuration:

```
log_config: "CONFDIR/SERVERNAME.log.config"
```

Ratelimiting

Options related to ratelimiting in Synapse.

Each ratelimiting configuration is made of two parameters:

- `per_second`: number of requests a client can send per second.
 - `burst_count`: number of requests a client can send before being throttled.
-

rc_message

Ratelimiting settings for client messaging.

This is a ratelimiting option for messages that ratelimits sending based on the account the client is using. It defaults to: `per_second: 0.2`, `burst_count: 10`.

Example configuration:

```
rc_message:  
  per_second: 0.5  
  burst_count: 15
```

rc_registration

This option ratelimits registration requests based on the client's IP address. It defaults to `per_second: 0.17`, `burst_count: 3`.

Example configuration:

```
rc_registration:  
  per_second: 0.15  
  burst_count: 2
```

rc_registration_token_validity

This option checks the validity of registration tokens that ratelimits requests based on the client's IP address. Defaults to `per_second: 0.1`, `burst_count: 5`.

Example configuration:

```
rc_registration_token_validity:  
  per_second: 0.3  
  burst_count: 6
```

rc_login

This option specifies several limits for login:

- `address` ratelimits login requests based on the client's IP address. Defaults to `per_second: 0.003, burst_count: 5`.
- `account` ratelimits login requests based on the account the client is attempting to log into. Defaults to `per_second: 0.003, burst_count: 5`.
- `failed_attempts` ratelimits login requests based on the account the client is attempting to log into, based on the amount of failed login attempts for this account. Defaults to `per_second: 0.17, burst_count: 3`.

Example configuration:

```
rc_login:
  address:
    per_second: 0.15
    burst_count: 5
  account:
    per_second: 0.18
    burst_count: 4
  failed_attempts:
    per_second: 0.19
    burst_count: 7
```

rc_admin_redaction

This option sets ratelimiting redactions by room admins. If this is not explicitly set then it uses the same ratelimiting as per `rc_message`. This is useful to allow room admins to deal with abuse quickly.

Example configuration:

```
rc_admin_redaction:
  per_second: 1
  burst_count: 50
```

rc_joins

This option allows for ratelimiting number of rooms a user can join. This setting has the following sub-options:

- `local`: ratelimits when users are joining rooms the server is already in. Defaults to `per_second: 0.1, burst_count: 10`.
- `remote`: ratelimits when users are trying to join rooms not on the server (which can be more computationally expensive than restricting locally). Defaults to `per_second: 0.01, burst_count: 10`

Example configuration:

```
rc_joins:
  local:
    per_second: 0.2
    burst_count: 15
  remote:
    per_second: 0.03
    burst_count: 12
```

`rc_joins_per_room`

This option allows admins to ratelimit joins to a room based on the number of recent joins (local or remote) to that room. It is intended to mitigate mass-join spam waves which target multiple homeservers.

By default, one join is permitted to a room every second, with an accumulating buffer of up to ten instantaneous joins.

Example configuration (default values):

```
rc_joins_per_room:
  per_second: 1
  burst_count: 10
```

Added in Synapse 1.64.0.

`rc_3pid_validation`

This option ratelimits how often a user or IP can attempt to validate a 3PID. Defaults to `per_second: 0.003, burst_count: 5`.

Example configuration:

```
rc_3pid_validation:
  per_second: 0.003
  burst_count: 5
```

rc_invites

This option sets ratelimiting how often invites can be sent in a room or to a specific user. `per_room` defaults to `per_second: 0.3`, `burst_count: 10`, `per_user` defaults to `per_second: 0.003`, `burst_count: 5`, and `per_issuer` defaults to `per_second: 0.3`, `burst_count: 10`.

Client requests that invite user(s) when [creating a room](#) will count against the `rc_invites.per_room` limit, whereas client requests to [invite a single user to a room](#) will count against both the `rc_invites.per_user` and `rc_invites.per_room` limits.

Federation requests to invite a user will count against the `rc_invites.per_user` limit only, as Synapse presumes ratelimiting by room will be done by the sending server.

The `rc_invites.per_user` limit applies to the *receiver* of the invite, rather than the sender, meaning that a `rc_invite.per_user.burst_count` of 5 mandates that a single user cannot *receive* more than a burst of 5 invites at a time.

In contrast, the `rc_invites.per_issuer` limit applies to the *issuer* of the invite, meaning that a `rc_invite.per_issuer.burst_count` of 5 mandates that single user cannot *send* more than a burst of 5 invites at a time.

Changed in version 1.63: added the `per_issuer` limit.

Example configuration:

```
rc_invites:
  per_room:
    per_second: 0.5
    burst_count: 5
  per_user:
    per_second: 0.004
    burst_count: 3
  per_issuer:
    per_second: 0.5
    burst_count: 5
```

rc_third_party_invite

This option ratelimits 3PID invites (i.e. invites sent to a third-party ID such as an email address or a phone number) based on the account that's sending the invite. Defaults to `per_second: 0.2`, `burst_count: 10`.

Example configuration:

```
rc_third_party_invite:  
  per_second: 0.2  
  burst_count: 10
```

rc_media_create

This option ratelimits creation of MXC URLs via the `/_matrix/media/v1/create` endpoint based on the account that's creating the media. Defaults to `per_second: 10`, `burst_count: 50`.

Example configuration:

```
rc_media_create:  
  per_second: 10  
  burst_count: 50
```

rc_federation

Defines limits on federation requests.

The `rc_federation` configuration has the following sub-options:

- `window_size`: window size in milliseconds. Defaults to 1000.
- `sleep_limit`: number of federation requests from a single server in a window before the server will delay processing the request. Defaults to 10.
- `sleep_delay`: duration in milliseconds to delay processing events from remote servers by if they go over the sleep limit. Defaults to 500.
- `reject_limit`: maximum number of concurrent federation requests allowed from a single server. Defaults to 50.
- `concurrent`: number of federation requests to concurrently process from a single server. Defaults to 3.

Example configuration:

```
rc_federation:  
  window_size: 750  
  sleep_limit: 15  
  sleep_delay: 400  
  reject_limit: 40  
  concurrent: 5
```

rc_presence

This option sets ratelimiting for presence.

The `rc_presence.per_user` option sets rate limits on how often a specific users' presence updates are evaluated. Ratelimited presence updates sent via sync are ignored, and no error is returned to the client. This option also sets the rate limit for the `PUT /_matrix/client/v3/presence/{userId}/status` endpoint.

`per_user` defaults to `per_second: 0.1, burst_count: 1`.

Example configuration:

```
rc_presence:  
  per_user:  
    per_second: 0.05  
    burst_count: 1
```

federation_rr_transactions_per_room_per_second

Sets outgoing federation transaction frequency for sending read-receipts, per-room.

If we end up trying to send out more read-receipts, they will get buffered up into fewer transactions. Defaults to 50.

Example configuration:

```
federation_rr_transactions_per_room_per_second: 40
```

Media Store

Config options related to Synapse's media store.

enable_authenticated_media

When set to true, all subsequent media uploads will be marked as authenticated, and will not be available over legacy unauthenticated media endpoints

`/_matrix/media/(r0|v3|v1)/download` and `/_matrix/media/(r0|v3|v1)/thumbnail` - requests for authenticated media over these endpoints will result in a 404. All media, including authenticated media, will be available over the authenticated media endpoints `_matrix/client/v1/media/download` and `_matrix/client/v1/media/thumbnail`. Media

uploaded prior to setting this option to true will still be available over the legacy endpoints. Note if the setting is switched to false after enabling, media marked as authenticated will be available over legacy endpoints. Defaults to true (previously false). In a future release of Synapse, this option will be removed and become always-on.

In all cases, authenticated requests to download media will succeed, but for unauthenticated requests, this case-by-case breakdown describes whether media downloads are permitted:

- `enable_authenticated_media = False`:
 - unauthenticated client or homeserver requesting local media: allowed
 - unauthenticated client or homeserver requesting remote media: allowed as long as the media is in the cache, or as long as the remote homeserver does not require authentication to retrieve the media
- `enable_authenticated_media = True`:
 - unauthenticated client or homeserver requesting local media: allowed if the media was stored on the server whilst `enable_authenticated_media` was `False` (or in a previous Synapse version where this option did not exist); otherwise denied.
 - unauthenticated client or homeserver requesting remote media: the same as for local media; allowed if the media was stored on the server whilst `enable_authenticated_media` was `False` (or in a previous Synapse version where this option did not exist); otherwise denied.

It is especially notable that media downloaded before this option existed (in older Synapse versions), or whilst this option was set to `False`, will perpetually be available over the legacy, unauthenticated endpoint, even after this option is set to `True`. This is for backwards compatibility with older clients and homeservers that do not yet support requesting authenticated media; those older clients or homeservers will not be cut off from media they can already see.

Changed in Synapse 1.120: This option now defaults to `True` when not set, whereas before this version it defaulted to `False`.

Example configuration:

```
enable_authenticated_media: false
```

`enable_media_repo`

Enable the media store service in the Synapse master. Defaults to true. Set to false if you are using a separate media store worker.

Example configuration:

```
enable_media_repo: false
```

media_store_path

Directory where uploaded images and attachments are stored.

Example configuration:

```
media_store_path: "DATADIR/media_store"
```

max_pending_media_uploads

How many *pending media uploads* can a given user have? A pending media upload is a created MXC URI that (a) is not expired (the `unused_expires_at` timestamp has not passed) and (b) the media has not yet been uploaded for. Defaults to 5.

Example configuration:

```
max_pending_media_uploads: 5
```

unused_expiration_time

How long to wait in milliseconds before expiring created media IDs. Defaults to "24h"

Example configuration:

```
unused_expiration_time: "1h"
```

media_storage_providers

Media storage providers allow media to be stored in different locations. Defaults to none. Associated sub-options are:

- `module`: type of resource, e.g. `file_system`.
- `store_local`: whether to store newly uploaded local files
- `store_remote`: whether to store newly downloaded local files
- `store_synchronous`: whether to wait for successful storage for local uploads
- `config`: sets a path to the resource through the `directory` option

Example configuration:

```
media_storage_providers:
  - module: file_system
    store_local: false
    store_remote: false
    store_synchronous: false
    config:
      directory: /mnt/some/other/directory
```

max_upload_size

The largest allowed upload size in bytes.

If you are using a reverse proxy you may also need to set this value in your reverse proxy's config. Defaults to 50M. Notably Nginx has a small max body size by default. See [here](#) for more on using a reverse proxy with Synapse.

Example configuration:

```
max_upload_size: 60M
```

max_image_pixels

Maximum number of pixels that will be thumbnailled. Defaults to 32M.

Example configuration:

```
max_image_pixels: 35M
```

remote_media_download_burst_count

Remote media downloads are ratelimited using a [leaky bucket algorithm](#), where a given "bucket" is keyed to the IP address of the requester when requesting remote media downloads. This configuration option sets the size of the bucket against which the size in bytes of downloads are penalized - if the bucket is full, ie a given number of bytes have already been downloaded, further downloads will be denied until the bucket drains. Defaults to 500MiB. See also `remote_media_download_per_second` which determines the rate at which the "bucket" is emptied and thus has available space to authorize new requests.

Example configuration:

```
remote_media_download_burst_count: 200M
```

remote_media_download_per_second

Works in conjunction with `remote_media_download_burst_count` to ratelimit remote media downloads - this configuration option determines the rate at which the "bucket" (see above) leaks in bytes per second. As requests are made to download remote media, the size of those requests in bytes is added to the bucket, and once the bucket has reached its capacity, no more requests will be allowed until a number of bytes has "drained" from the bucket. This setting determines the rate at which bytes drain from the bucket, with the practical effect that the larger the number, the faster the bucket leaks, allowing for more bytes downloaded over a shorter period of time. Defaults to 87KiB per second. See also `remote_media_download_burst_count`.

Example configuration:

```
remote_media_download_per_second: 40K
```

prevent_media_downloads_from

A list of domains to never download media from. Media from these domains that is already downloaded will not be deleted, but will be inaccessible to users. This option does not affect admin APIs trying to download/operate on media.

This will not prevent the listed domains from accessing media themselves. It simply prevents users on this server from downloading media originating from the listed servers.

This will have no effect on media originating from the local server. This only affects media downloaded from other Matrix servers, to control URL previews see `url_preview_ip_range_blacklist` or `url_preview_url_blacklist`.

Defaults to an empty list (nothing blocked).

Example configuration:

```
prevent_media_downloads_from:
  - evil.example.org
  - evil2.example.org
```

dynamic_thumbnails

Whether to generate new thumbnails on the fly to precisely match the resolution requested by the client. If true then whenever a new resolution is requested by the client the server will generate a new thumbnail. If false the server will pick a thumbnail from a precalculated list. Defaults to false.

Example configuration:

```
dynamic_thumbnails: true
```

thumbnail_sizes

List of thumbnails to precalculate when an image is uploaded. Associated sub-options are:

- `width`
- `height`
- `method`: i.e. `crop`, `scale`, etc.

Example configuration:

```
thumbnail_sizes:
- width: 32
  height: 32
  method: crop
- width: 96
  height: 96
  method: crop
- width: 320
  height: 240
  method: scale
- width: 640
  height: 480
  method: scale
- width: 800
  height: 600
  method: scale
```

media_retention

Controls whether local media and entries in the remote media cache (media that is downloaded from other homeservers) should be removed under certain conditions, typically for the purpose of saving space.

Purging media files will be carried out by the media worker (that is, the worker that has the `enable_media_repo` homeserver config option set to 'true'). This may be the main

process.

The `media_retention.local_media_lifetime` and `media_retention.remote_media_lifetime` config options control whether media will be purged if it has not been accessed in a given amount of time. Note that media is 'accessed' when loaded in a room in a client, or otherwise downloaded by a local or remote user. If the media has never been accessed, the media's creation time is used instead. Both thumbnails and the original media will be removed. If either of these options are unset, then media of that type will not be purged.

Local or cached remote media that has been [quarantined](#) will not be deleted. Similarly, local media that has been marked as [protected from quarantine](#) will not be deleted.

Example configuration:

```
media_retention:  
  local_media_lifetime: 90d  
  remote_media_lifetime: 14d
```

`url_preview_enabled`

This setting determines whether the preview URL API is enabled. It is disabled by default. Set to true to enable. If enabled you must specify a `url_preview_ip_range_blacklist` blacklist.

Example configuration:

```
url_preview_enabled: true
```

`url_preview_ip_range_blacklist`

List of IP address CIDR ranges that the URL preview spider is denied from accessing. There are no defaults: you must explicitly specify a list for URL previewing to work. You should specify any internal services in your network that you do not want synapse to try to connect to, otherwise anyone in any Matrix room could cause your synapse to issue arbitrary GET requests to your internal services, causing serious security issues.

(0.0.0.0 and :: are always blacklisted, whether or not they are explicitly listed here, since they correspond to unroutable addresses.)

This must be specified if `url_preview_enabled` is set. It is recommended that you use the following example list as a starting point.

Note: The value is ignored when an HTTP proxy is in use.

Example configuration:

```
url_preview_ip_range_blacklist:
- '127.0.0.0/8'
- '10.0.0.0/8'
- '172.16.0.0/12'
- '192.168.0.0/16'
- '100.64.0.0/10'
- '192.0.0.0/24'
- '169.254.0.0/16'
- '192.88.99.0/24'
- '198.18.0.0/15'
- '192.0.2.0/24'
- '198.51.100.0/24'
- '203.0.113.0/24'
- '224.0.0.0/4'
- '::1/128'
- 'fe80::/10'
- 'fc00::/7'
- '2001:db8::/32'
- 'ff00::/8'
- 'fec0::/10'
```

url_preview_ip_range_whitelist

This option sets a list of IP address CIDR ranges that the URL preview spider is allowed to access even if they are specified in `url_preview_ip_range_blacklist`. This is useful for specifying exceptions to wide-ranging blacklisted target IP ranges - e.g. for enabling URL previews for a specific private website only visible in your network. Defaults to none.

Example configuration:

```
url_preview_ip_range_whitelist:
- '192.168.1.1'
```

url_preview_url_blacklist

Optional list of URL matches that the URL preview spider is denied from accessing. This is a usability feature, not a security one. You should use `url_preview_ip_range_blacklist` in preference to this, otherwise someone could define a public DNS entry that points to a private IP address and circumvent the blacklist. Applications that perform redirects or serve different content when detecting that Synapse is accessing them can also bypass the blacklist. This is more useful if you know there is an entire shape of URL that you know that you do not want Synapse to preview.

Each list entry is a dictionary of url component attributes as returned by urlparse.urlsplit as applied to the absolute form of the URL. See [here](#) for more information. Some examples are:

- `username`
- `netloc`
- `scheme`
- `path`

The values of the dictionary are treated as a filename match pattern applied to that component of URLs, unless they start with a ^ in which case they are treated as a regular expression match. If all the specified component matches for a given list item succeed, the URL is blacklisted.

Example configuration:

```
url_preview_url_blacklist:
  # blacklist any URL with a username in its URI
  - username: '.*'

  # blacklist all *.google.com URLs
  - netloc: 'google.com'
  - netloc: '*.google.com'

  # blacklist all plain HTTP URLs
  - scheme: 'http'

  # blacklist http(s)://www.acme.com/fo
  - netloc: 'www.acme.com'
  path: '/fo'

  # blacklist any URL with a literal IPv4 address
  - netloc: '^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$'
```

max_spider_size

The largest allowed URL preview spidering size in bytes. Defaults to 10M.

Example configuration:

```
max_spider_size: 8M
```

url_preview_accept_language

A list of values for the Accept-Language HTTP header used when downloading webpages during URL preview generation. This allows Synapse to specify the preferred languages that

URL previews should be in when communicating with remote servers.

Each value is a IETF language tag; a 2-3 letter identifier for a language, optionally followed by subtags separated by '-', specifying a country or region variant.

Multiple values can be provided, and a weight can be added to each by using quality value syntax (`;q=`). '*' translates to any language.

Defaults to "en".

Example configuration:

```
url_preview_accept_language:
  - 'en-UK'
  - 'en-US;q=0.9'
  - 'fr;q=0.8'
  - '*;q=0.7'
```

oembed

oEmbed allows for easier embedding content from a website. It can be used for generating URLs previews of services which support it. A default list of oEmbed providers is included with Synapse. Set `disable_default_providers` to true to disable using these default oEmbed URLs. Use `additional_providers` to specify additional files with oEmbed configuration (each should be in the form of providers.json). By default this list is empty.

Example configuration:

```
oembed:
  disable_default_providers: true
  additional_providers:
    - oembed/my_providers.json
```

Captcha

See [here](#) for full details on setting up captcha.

recaptcha_public_key

This homeserver's ReCAPTCHA public key. Must be specified if `enable_registration_captcha` is enabled.

Example configuration:

```
recaptcha_public_key: "YOUR_PUBLIC_KEY"
```

recaptcha_private_key

This homeserver's ReCAPTCHA private key. Must be specified if `enable_registration_captcha` is enabled.

Example configuration:

```
recaptcha_private_key: "YOUR_PRIVATE_KEY"
```

enable_registration_captcha

Set to `true` to require users to complete a CAPTCHA test when registering an account. Requires a valid ReCaptcha public/private key. Defaults to `false`.

Note that `enable_registration` must also be set to allow account registration.

Example configuration:

```
enable_registration_captcha: true
```

recaptcha_siteverify_api

The API endpoint to use for verifying `m.login.recaptcha` responses. Defaults to <https://www.recaptcha.net/recaptcha/api/siteverify>.

Example configuration:

```
recaptcha_siteverify_api: "https://my.recaptcha.site"
```

TURN

Options related to adding a TURN server to Synapse.

turn_uris

The public URIs of the TURN server to give to clients.

Example configuration:

```
turn_uris: [turn:example.org]
```

turn_shared_secret

The shared secret used to compute passwords for the TURN server.

Example configuration:

```
turn_shared_secret: "YOUR_SHARED_SECRET"
```

turn_shared_secret_path

An alternative to `turn_shared_secret`: allows the shared secret to be specified in an external file.

The file should be a plain text file, containing only the shared secret. Synapse reads the shared secret from the given file once at startup.

Example configuration:

```
turn_shared_secret_path: /path/to/secrets/file
```

Added in Synapse 1.116.0.

turn_username and turn_password

The Username and password if the TURN server needs them and does not use a token.

Example configuration:

```
turn_username: "TURN SERVER USERNAME"  
turn_password: "TURN SERVER PASSWORD"
```

turn_user_lifetime

How long generated TURN credentials last. Defaults to 1h.

Example configuration:

```
turn_user_lifetime: 2h
```

turn_allow_guests

Whether guests should be allowed to use the TURN server. This defaults to true, otherwise VoIP will be unreliable for guests. However, it does introduce a slight security risk as it allows users to connect to arbitrary endpoints without having first signed up for a valid account (e.g. by passing a CAPTCHA).

Example configuration:

```
turn_allow_guests: false
```

Registration

Registration can be rate-limited using the parameters in the [Ratelimiting](#) section of this manual.

enable_registration

Enable registration for new users. Defaults to `false`.

It is highly recommended that if you enable registration, you set one or more of the following options, to avoid abuse of your server by "bots":

- `enable_registration_captcha`
- `registrations_require_3pid`
- `registration_requires_token`

(In order to enable registration without any verification, you must also set `enable_registration_without_verification`.)

Note that even if this setting is disabled, new accounts can still be created via the admin API if `registration_shared_secret` is set.

Example configuration:

```
enable_registration: true
```

enable_registration_without_verification

Enable registration without email or captcha verification. Note: this option is *not* recommended, as registration without verification is a known vector for spam and abuse. Defaults to `false`. Has no effect unless `enable_registration` is also enabled.

Example configuration:

```
enable_registration_without_verification: true
```

registrations_require_3pid

If this is set, users must provide all of the specified types of **3PID** when registering an account.

Note that `enable_registration` must also be set to allow account registration.

Example configuration:

```
registrations_require_3pid:
  - email
  - msisdn
```

disable_msisdn_registration

Explicitly disable asking for MSISDNs from the registration flow (overrides `registrations_require_3pid` if MSISDNs are set as required).

Example configuration:

```
disable_msisdn_registration: true
```

allowed_local_3pids

Mandate that users are only allowed to associate certain formats of 3PIDs with accounts on this server, as specified by the `medium` and `pattern` sub-options. `pattern` is a Perl-like regular expression.

More information about 3PIDs, allowed `medium` types and their `address` syntax can be found [in the Matrix spec](#).

Example configuration:

```
allowed_local_3pids:
  - medium: email
    pattern: '^[^@]+@matrix\.org$'
  - medium: email
    pattern: '^[^@]+@vector\.im$'
  - medium: msisdn
    pattern: '^44\d{10}$'
```

enable_3pid_lookup

Enable 3PIDs lookup requests to identity servers from this server. Defaults to true.

Example configuration:

```
enable_3pid_lookup: false
```

registration_requires_token

Require users to submit a token during registration. Tokens can be managed using the admin [API](#). Disabling this option will not delete any tokens previously generated. Defaults to `false`. Set to `true` to enable.

Note that `enable_registration` must also be set to allow account registration.

Example configuration:

```
registration_requires_token: true
```

registration_shared_secret

If set, allows registration of standard or admin accounts by anyone who has the shared secret, even if `enable_registration` is not set.

This is primarily intended for use with the `register_new_matrix_user` script (see [Registering a user](#)); however, the interface is [documented](#).

Replacing an existing `registration_shared_secret` with a new one requires users of the [Shared-Secret Registration API](#) to start using the new secret for requesting any further one-time nonces.

⚠ Warning – The additional consequences of replacing `macaroon_secret_key` will apply in case it delegates to `registration_shared_secret`.

See also `registration_shared_secret_path`.

Example configuration:

```
registration_shared_secret: <PRIVATE STRING>
```

registration_shared_secret_path

An alternative to `registration_shared_secret`: allows the shared secret to be specified in an external file.

The file should be a plain text file, containing only the shared secret.

If this file does not exist, Synapse will create a new shared secret on startup and store it in this file.

Example configuration:

```
registration_shared_secret_path: /path/to/secrets/file
```

Added in Synapse 1.67.0.

bcrypt_rounds

Set the number of bcrypt rounds used to generate password hash. Larger numbers increase the work factor needed to generate the hash. The default number is 12 (which equates to

2^{12} rounds). N.B. that increasing this will exponentially increase the time required to register or login - e.g. 24 => 2^{24} rounds which will take >20 mins. Example configuration:

```
bcrypt_rounds: 14
```

allow_guest_access

Allows users to register as guests without a password/email/etc, and participate in rooms hosted on this server which have been made accessible to anonymous users. Defaults to false.

Example configuration:

```
allow_guest_access: true
```

default_identity_server

The identity server which we suggest that clients should use when users log in on this server.

(By default, no suggestion is made, so it is left up to the client. This setting is ignored unless `public_baseurl` is also explicitly set.)

Example configuration:

```
default_identity_server: https://matrix.org
```

account_threepid_delegates

Delegate verification of phone numbers to an identity server.

When a user wishes to add a phone number to their account, we need to verify that they actually own that phone number, which requires sending them a text message (SMS). Currently Synapse does not support sending those texts itself and instead delegates the task to an identity server. The base URI for the identity server to be used is specified by the `account_threepid_delegates.msisdn` option.

If this is left unspecified, Synapse will not allow users to add phone numbers to their account.

(Servers handling these requests must answer the `/requestToken` endpoints defined by the Matrix Identity Service API specification.)

Deprecated in Synapse 1.64.0: The `email` option is deprecated.

Removed in Synapse 1.66.0: The `email` option has been removed. If present, Synapse will report a configuration error on startup.

Example configuration:

```
account_threepid_delegates:  
  msisdn: http://localhost:8090 # Delegate SMS sending to this local process
```

enable_set_displayname

Whether users are allowed to change their displayname after it has been initially set. Useful when provisioning users based on the contents of a third-party directory.

Does not apply to server administrators. Defaults to true.

Example configuration:

```
enable_set_displayname: false
```

enable_set_avatar_url

Whether users are allowed to change their avatar after it has been initially set. Useful when provisioning users based on the contents of a third-party directory.

Does not apply to server administrators. Defaults to true.

Example configuration:

```
enable_set_avatar_url: false
```

enable_3pid_changes

Whether users can change the third-party IDs associated with their accounts (email address and msisdn).

Defaults to true.

Example configuration:

```
enable_3pid_changes: false
```

auto_join_rooms

Users who register on this homeserver will automatically be joined to the rooms listed under this option.

By default, any room aliases included in this list will be created as a publicly joinable room when the first user registers for the homeserver. If the room already exists, make certain it is a publicly joinable room, i.e. the join rule of the room must be set to 'public'. You can find more options relating to auto-joining rooms below.

As Spaces are just rooms under the hood, Space aliases may also be used.

Example configuration:

```
auto_join_rooms:
- "#exampleroom:example.com"
- "#anotherexampleroom:example.com"
```

autocreate_auto_join_rooms

Where `auto_join_rooms` are specified, setting this flag ensures that the rooms exist by creating them when the first user on the homeserver registers. This option will not create Spaces.

By default the auto-created rooms are publicly joinable from any federated server. Use the `autocreate_auto_join_rooms_federated` and `autocreate_auto_join_room_preset` settings to customise this behaviour.

Setting to false means that if the rooms are not manually created, users cannot be auto-joined since they do not exist.

Defaults to true.

Example configuration:

```
autocreate_auto_join_rooms: false
```

autocreate_auto_join_rooms_federated

Whether the rooms listed in `auto_join_rooms` that are auto-created are available via federation. Only has an effect if `autocreate_auto_join_rooms` is true.

Note that whether a room is federated cannot be modified after creation.

Defaults to true: the room will be joinable from other servers. Set to false to prevent users from other homeservers from joining these rooms.

Example configuration:

```
autocreate_auto_join_rooms_federated: false
```

autocreate_auto_join_room_preset

The room preset to use when auto-creating one of `auto_join_rooms`. Only has an effect if `autocreate_auto_join_rooms` is true.

Possible values for this option are:

- "public_chat": the room is joinable by anyone, including federated servers if `autocreate_auto_join_rooms_federated` is true (the default).
- "private_chat": an invitation is required to join these rooms.
- "trusted_private_chat": an invitation is required to join this room and the invitee is assigned a power level of 100 upon joining the room.

Each preset will set up a room in the same manner as if it were provided as the `preset` parameter when calling the [POST /_matrix/client/v3/createRoom](#) Client-Server API endpoint.

If a value of "private_chat" or "trusted_private_chat" is used then `auto_join_mxid_localpart` must also be configured.

Defaults to "public_chat".

Example configuration:

```
autocreate_auto_join_room_preset: private_chat
```

auto_join_mxid_localpart

The local part of the user id which is used to create `auto_join_rooms` if `autocreate_auto_join_rooms` is true. If this is not provided then the initial user account

that registers will be used to create the rooms.

The user id is also used to invite new users to any auto-join rooms which are set to invite-only.

It *must* be configured if `autocreate_auto_join_room_preset` is set to "private_chat" or "trusted_private_chat".

Note that this must be specified in order for new users to be correctly invited to any auto-join rooms which have been set to invite-only (either at the time of creation or subsequently).

Note that, if the room already exists, this user must be joined and have the appropriate permissions to invite new members.

Example configuration:

```
auto_join_mxid_localpart: system
```

auto_join_rooms_for_guests

When `auto_join_rooms` is specified, setting this flag to false prevents guest accounts from being automatically joined to the rooms.

Defaults to true.

Example configuration:

```
auto_join_rooms_for_guests: false
```

inhibit_user_in_use_error

Whether to inhibit errors raised when registering a new account if the user ID already exists. If turned on, requests to `/register/available` will always show a user ID as available, and Synapse won't raise an error when starting a registration with a user ID that already exists. However, Synapse will still raise an error if the registration completes and the username conflicts.

Defaults to false.

Example configuration:

```
inhibit_user_in_use_error: true
```

User session management

session_lifetime

Time that a user's session remains valid for, after they log in.

Note that this is not currently compatible with guest logins.

Note also that this is calculated at login time: changes are not applied retrospectively to users who have already logged in.

By default, this is infinite.

Example configuration:

```
session_lifetime: 24h
```

refreshable_access_token_lifetime

Time that an access token remains valid for, if the session is using refresh tokens.

For more information about refresh tokens, please see the [manual](#).

Note that this only applies to clients which advertise support for refresh tokens.

Note also that this is calculated at login time and refresh time: changes are not applied to existing sessions until they are refreshed.

By default, this is 5 minutes.

Example configuration:

```
refreshable_access_token_lifetime: 10m
```

refresh_token_lifetime

Time that a refresh token remains valid for (provided that it is not exchanged for another one first). This option can be used to automatically log-out inactive sessions. Please see the [manual](#) for more information.

Note also that this is calculated at login time and refresh time: changes are not applied to existing sessions until they are refreshed.

By default, this is infinite.

Example configuration:

```
refresh_token_lifetime: 24h
```

nonrefreshable_access_token_lifetime

Time that an access token remains valid for, if the session is NOT using refresh tokens.

Please note that not all clients support refresh tokens, so setting this to a short value may be inconvenient for some users who will then be logged out frequently.

Note also that this is calculated at login time: changes are not applied retrospectively to existing sessions for users that have already logged in.

By default, this is infinite.

Example configuration:

```
nonrefreshable_access_token_lifetime: 24h
```

ui_auth

The amount of time to allow a user-interactive authentication session to be active.

This defaults to 0, meaning the user is queried for their credentials before every action, but this can be overridden to allow a single validation to be re-used. This weakens the protections afforded by the user-interactive authentication process, by allowing for multiple (and potentially different) operations to use the same validation session.

This is ignored for potentially "dangerous" operations (including deactivating an account, modifying an account password, adding a 3PID, and minting additional login tokens).

Use the `session_timeout` sub-option here to change the time allowed for credential validation.

Example configuration:

```
ui_auth:
  session_timeout: "15s"
```

login_via_existing_session

Matrix supports the ability of an existing session to mint a login token for another client.

Synapse disables this by default as it has security ramifications -- a malicious client could use the mechanism to spawn more than one session.

The duration of time the generated token is valid for can be configured with the `token_timeout` sub-option.

User-interactive authentication is required when this is enabled unless the `require_ui_auth` sub-option is set to `False`.

Example configuration:

```
login_via_existing_session:
  enabled: true
  require_ui_auth: false
  token_timeout: "5m"
```

Metrics

Config options related to metrics.

enable_metrics

Set to true to enable collection and rendering of performance metrics. Defaults to false.

Example configuration:

```
enable_metrics: true
```

sentry

Use this option to enable sentry integration. Provide the DSN assigned to you by sentry with the `dsn` setting.

An optional `environment` field can be used to specify an environment. This allows for log maintenance based on different environments, ensuring better organization and analysis..

NOTE: While attempts are made to ensure that the logs don't contain any sensitive information, this cannot be guaranteed. By enabling this option the sentry server may therefore receive sensitive information, and it in turn may then disseminate sensitive information through insecure notification channels if so configured.

Example configuration:

```
sentry:  
  environment: "production"  
  dsn: "..."
```

metrics_flags

Flags to enable Prometheus metrics which are not suitable to be enabled by default, either for performance reasons or limited use. Currently the only option is `known_servers`, which publishes `synapse_federation_known_servers`, a gauge of the number of servers this homeserver knows about, including itself. May cause performance problems on large homeservers.

Example configuration:

```
metrics_flags:  
  known_servers: true
```

report_stats

Whether or not to report homeserver usage statistics. This is originally set when generating the config. Set this option to true or false to change the current behavior. See [Reporting Homeserver Usage Statistics](#) for information on what data is reported.

Statistics will be reported 5 minutes after Synapse starts, and then every 3 hours after that.

Example configuration:

```
report_stats: true
```

report_stats_endpoint

The endpoint to report homeserver usage statistics to. Defaults to `https://matrix.org/report-usage-stats/push`

Example configuration:

```
report_stats_endpoint: https://example.com/report-usage-stats/push
```

API Configuration

Config settings related to the client/server API

room_prejoin_state

This setting controls the state that is shared with users upon receiving an invite to a room, or in reply to a knock on a room. By default, the following state events are shared with users:

- `m.room.join_rules`
- `m.room.canonical_alias`
- `m.room.avatar`
- `m.room.encryption`
- `m.room.name`
- `m.room.create`
- `m.room.topic`

To change the default behavior, use the following sub-options:

- `disable_default_event_types`: boolean. Set to `true` to disable the above defaults. If this is enabled, only the event types listed in `additional_event_types` are shared. Defaults to `false`.
- `additional_event_types`: A list of additional state events to include in the events to be shared. By default, this list is empty (so only the default event types are shared).

Each entry in this list should be either a single string or a list of two strings.

- A standalone string `t` represents all events with type `t` (i.e. with no restrictions on state keys).
- A pair of strings `[t, s]` represents a single event with type `t` and state key `s`. The same type can appear in two entries with different state keys: in this situation, both state keys are included in prejoin state.

Example configuration:

```

room_prejoin_state:
  disable_default_event_types: false
  additional_event_types:
    # Share all events of type `org.example.custom.event.typeA`
    - org.example.custom.event.typeA
    # Share only events of type `org.example.custom.event.typeB` whose
    # state_key is "foo"
    - ["org.example.custom.event.typeB", "foo"]
    # Share only events of type `org.example.custom.event.typeC` whose
    # state_key is "bar" or "baz"
    - ["org.example.custom.event.typeC", "bar"]
    - ["org.example.custom.event.typeC", "baz"]

```

Changed in Synapse 1.74: admins can filter the events in prejoin state based on their state key.

track_puppeted_user_ips

We record the IP address of clients used to access the API for various reasons, including displaying it to the user in the "Where you're signed in" dialog.

By default, when puppeting another user via the admin API, the client IP address is recorded against the user who created the access token (ie, the admin user), and *not* the puppeted user.

Set this option to true to also record the IP address against the puppeted user. (This also means that the puppeted user will count as an "active" user for the purpose of monthly active user tracking - see `limit_usage_by_mau` etc above.)

Example configuration:

```
track_puppeted_user_ips: true
```

app_service_config_files

A list of application service config files to use.

Example configuration:

```

app_service_config_files:
  - app_service_1.yaml
  - app_service_2.yaml

```

track_appservice_user_ips

Defaults to false. Set to true to enable tracking of application service IP addresses. Implicitly enables MAU tracking for application service users.

Example configuration:

```
track_appservice_user_ips: true
```

use_appservice_legacy_authorization

Whether to send the application service access tokens via the `access_token` query parameter per older versions of the Matrix specification. Defaults to false. Set to true to enable sending access tokens via a query parameter.

**Enabling this option is considered insecure and is not recommended. **

Example configuration:

```
use_appservice_legacy_authorization: true
```

macaroon_secret_key

A secret which is used to sign

- access token for guest users,
- short-term login token used during SSO logins (OIDC or SAML2) and
- token used for unsubscribing from email notifications.

If none is specified, the `registration_shared_secret` is used, if one is given; otherwise, a secret key is derived from the signing key.

⚠ Warning – Replacing an existing `macaroon_secret_key` with a new one will lead to invalidation of access tokens for all guest users. It will also break unsubscribe links in emails sent before the change. An unlucky user might encounter a broken SSO login flow and would have to start again.

Example configuration:

```
macaroon_secret_key: <PRIVATE STRING>
```

macaroon_secret_key_path

An alternative to `macaroon_secret_key`: allows the secret key to be specified in an external file.

The file should be a plain text file, containing only the secret key. Synapse reads the secret key from the given file once at startup.

Example configuration:

```
macaroon_secret_key_path: /path/to/secrets/file
```

Added in Synapse 1.121.0.

form_secret

A secret which is used to calculate HMACs for form values, to stop falsification of values. Must be specified for the User Consent forms to work.

Replacing an existing `form_secret` with a new one might break the user consent page for an unlucky user and require them to reopen the page from a new link.

Example configuration:

```
form_secret: <PRIVATE STRING>
```

Signing Keys

Config options relating to signing keys

signing_key_path

Path to the signing key to sign events and federation requests with.

New in Synapse 1.67: If this file does not exist, Synapse will create a new signing key on startup and store it in this file.

Example configuration:

```
signing_key_path: "CONFDIR/SERVERNAME.signing.key"
```

old_signing_keys

The keys that the server used to sign messages with but won't use to sign new messages. For each key, `key` should be the base64-encoded public key, and `expired_ts` should be the time (in milliseconds since the unix epoch) that it was last used.

It is possible to build an entry from an old `signing.key` file using the `export_signing_key` script which is provided with synapse.

If you have lost the private key file, you can ask another server you trust to tell you the public keys it has seen from your server. To fetch the keys from `matrix.org`, try something like:

```
curl https://matrix-federation.matrix.org/_matrix/key/v2/query/myserver.example.com | jq '.server_keys | map(.verify_keys) | add'
```

Example configuration:

```
old_signing_keys:
  "ed25519:id": { key: "base64string", expired_ts: 123456789123 }
```

key_refresh_interval

How long key response published by this server is valid for. Used to set the `valid_until_ts` in `/key/v2` APIs. Determines how quickly servers will query to check which keys are still valid. Defaults to 1d.

Example configuration:

```
key_refresh_interval: 2d
```

trusted_key_servers

The trusted servers to download signing keys from.

When we need to fetch a signing key, each server is tried in parallel.

Normally, the connection to the key server is validated via TLS certificates. Additional security can be provided by configuring a `verify_key`, which will make synapse check that the response is signed by that key.

This setting supersedes an older setting named `perspectives`. The old format is still supported for backwards-compatibility, but it is deprecated.

`trusted_key_servers` defaults to `matrix.org`, but using it will generate a warning on start-up. To suppress this warning, set `suppress_key_server_warning` to `true`.

If the use of a trusted key server has to be deactivated, e.g. in a private federation or for privacy reasons, this can be realised by setting an empty array (`trusted_key_servers: []`). Then Synapse will request the keys directly from the server that owns the keys. If Synapse does not get keys directly from the server, the events of this server will be rejected.

Options for each entry in the list include:

- `server_name`: the name of the server. Required.
- `verify_keys`: an optional map from key id to base64-encoded public key. If specified, we will check that the response is signed by at least one of the given keys.
- `accept_keys_insecurely`: a boolean. Normally, if `verify_keys` is unset, and `federation_verify_certificates` is not `true`, synapse will refuse to start, because this would allow anyone who can spoof DNS responses to masquerade as the trusted key server. If you know what you are doing and are sure that your network environment provides a secure connection to the key server, you can set this to `true` to override this behaviour.

Example configuration #1:

```
trusted_key_servers:
  - server_name: "my_trusted_server.example.com"
    verify_keys:
      "ed25519:auto": "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"
  - server_name: "my_other_trusted_server.example.com"
```

Example configuration #2:

```
trusted_key_servers:
  - server_name: "matrix.org"
```

suppress_key_server_warning

Set the following to `true` to disable the warning that is emitted when the `trusted_key_servers` include '`matrix.org`'. See above.

Example configuration:

```
suppress_key_server_warning: true
```

key_server_signing_keys_path

The signing keys to use when acting as a trusted key server. If not specified defaults to the server signing key.

Can contain multiple keys, one per line.

Example configuration:

```
key_server_signing_keys_path: "key_server_signing_keys.key"
```

Single sign-on integration

The following settings can be used to make Synapse use a single sign-on provider for authentication, instead of its internal password database.

You will probably also want to set the following options to `false` to disable the regular login/registration flows:

- `enable_registration`
- `password_config.enabled`

saml2_config

Enable SAML2 for registration and login. Uses pysaml2. To learn more about pysaml and to find a full list options for configuring pysaml, read the docs [here](#).

At least one of `sp_config` or `config_path` must be set in this section to enable SAML login. You can either put your entire pysaml config inline using the `sp_config` option, or you can specify a path to a psyaml config file with the sub-option `config_path`. This setting has the following sub-options:

- `idp_name`: A user-facing name for this identity provider, which is used to offer the user a choice of login mechanisms.
- `idp_icon`: An optional icon for this identity provider, which is presented by clients and Synapse's own IdP picker page. If given, must be an MXC URI of the format `mx: //<server-name>/<media-id>`. (An easy way to obtain such an MXC URI is to upload an image to an (unencrypted) room and then copy the "url" from the source of the event.)
- `idp_brand`: An optional brand for this identity provider, allowing clients to style the login flow according to the identity provider in question. See the [spec](#) for possible options [here](#).

- **sp_config**: the configuration for the pysaml2 Service Provider. See pysaml2 docs for format of config. Default values will be used for the `entityid` and `service` settings, so it is not normally necessary to specify them unless you need to override them. Here are a few useful sub-options for configuring pysaml:
 - `metadata`: Point this to the IdP's metadata. You must provide either a local file via the `local` attribute or (preferably) a URL via the `remote` attribute.
 - `accepted_time_diff: 3`: Allowed clock difference in seconds between the homeserver and IdP. Defaults to 0.
 - `service`: By default, the user has to go to our login page first. If you'd like to allow IdP-initiated login, set `allow_unsolicited` to true under `sp` in the `service` section.
- **config_path**: specify a separate pysaml2 configuration file thusly: `config_path: "CONFDIR/sp_conf.py"`
- **saml_session_lifetime**: The lifetime of a SAML session. This defines how long a user has to complete the authentication process, if `allow_unsolicited` is unset. The default is 15 minutes.
- **user_mapping_provider**: Using this option, an external module can be provided as a custom solution to mapping attributes returned from a saml provider onto a matrix user. The `user_mapping_provider` has the following attributes:
 - `module`: The custom module's class.
 - `config`: Custom configuration values for the module. Use the values provided in the example if you are using the built-in `user_mapping_provider`, or provide your own config values for a custom class if you are using one. This section will be passed as a Python dictionary to the module's `parse_config` method. The built-in provider takes the following two options:
 - `mxid_source_attribute`: The SAML attribute (after mapping via the attribute maps) to use to derive the Matrix ID from. It is 'uid' by default. Note: This used to be configured by the `saml2_config.mxid_source_attribute option`. If that is still defined, its value will be used instead.
 - `mxid_mapping`: The mapping system to use for mapping the saml attribute onto a matrix ID. Options include: `hexencode` (which maps unpermitted characters to '=xx') and `dotreplace` (which replaces unpermitted characters with '.'). The default is `hexencode`. Note: This used to be configured by the `saml2_config.mxid_mapping option`. If that is still defined, its value will be used instead.
- **grandfathered_mxid_source_attribute**: In previous versions of synapse, the mapping from SAML attribute to MXID was always calculated dynamically rather than stored in a table. For backwards- compatibility, we will look for `user_ids` matching such a pattern before creating a new account. This setting controls the SAML attribute which will be used for this backwards-compatibility lookup. Typically it should be 'uid', but if the attribute maps are changed, it may be necessary to change it. The default is 'uid'.
- **attribute_requirements**: It is possible to configure Synapse to only allow logins if SAML attributes match particular values. The requirements can be listed under

- `attribute_requirements` as shown in the example. All of the listed attributes must match for the login to be permitted. Values can be specified in a `one_of` list to allow multiple values for an attribute.
- `idp_entityid`: If the metadata XML contains multiple IdP entities then the `idp_entityid` option must be set to the entity to redirect users to. Most deployments only have a single IdP entity and so should omit this option.

Once SAML support is enabled, a metadata file will be exposed at `https://<server>:<port>/_synapse/client/saml2/metadata.xml`, which you may be able to use to configure your SAML IdP with. Alternatively, you can manually configure the IdP to use an ACS location of `https://<server>:<port>/_synapse/client/saml2/authn_response`.

Example configuration:

```
saml2_config:
  sp_config:
    metadata:
      local: ["saml2/idp.xml"]
      remote:
        - url: https://our_idp/metadata.xml
  accepted_time_diff: 3

  service:
    sp:
      allow_unsolicited: true

  # The examples below are just used to generate our metadata xml, and you
  # may well not need them, depending on your setup. Alternatively you
  # may need a whole lot more detail - see the pysaml2 docs!
  description: ["My awesome SP", "en"]
  name: ["Test SP", "en"]

  ui_info:
    display_name:
      - lang: en
        text: "Display Name is the descriptive name of your service."
    description:
      - lang: en
        text: "Description should be a short paragraph explaining the purpose
of the service."
    information_url:
      - lang: en
        text: "https://example.com/terms-of-service"
    privacy_statement_url:
      - lang: en
        text: "https://example.com/privacy-policy"
    keywords:
      - lang: en
        text: ["Matrix", "Element"]
    logo:
      - lang: en
        text: "https://example.com/logo.svg"
        width: "200"
        height: "80"

  organization:
    name: Example com
    display_name:
      - ["Example co", "en"]
    url: "http://example.com"

  contact_person:
    - given_name: Bob
      sur_name: "the Sysadmin"
      email_address: ["admin@example.com"]
      contact_type: technical

  saml_session_lifetime: 5m

  user_mapping_provider:
    # Below options are intended for the built-in provider, they should be
```

```

# changed if using a custom module.
config:
  mxid_source_attribute: displayName
  mxid_mapping: dotreplace

grandfathered_mxid_source_attribute: upn

attribute_requirements:
  - attribute: userGroup
    value: "staff"
  - attribute: department
    one_of:
      - "sales"
      - "admins"

idp_entityid: 'https://our_idp/entityid'

```

oidc_providers

List of OpenID Connect (OIDC) / OAuth 2.0 identity providers, for registration and login. See [here](#) for information on how to configure these options.

For backwards compatibility, it is also possible to configure a single OIDC provider via an `oidc_config` setting. This is now deprecated and admins are advised to migrate to the `oidc_providers` format. (When doing that migration, use `oidc` for the `idp_id` to ensure that existing users continue to be recognised.)

Options for each entry include:

- `idp_id`: a unique identifier for this identity provider. Used internally by Synapse; should be a single word such as 'github'. Note that, if this is changed, users authenticating via that provider will no longer be recognised as the same user! (Use "oidc" here if you are migrating from an old `oidc_config` configuration.)
- `idp_name`: A user-facing name for this identity provider, which is used to offer the user a choice of login mechanisms.
- `idp_icon`: An optional icon for this identity provider, which is presented by clients and Synapse's own IdP picker page. If given, must be an MXC URI of the format `mxc://<server-name>/<media-id>`. (An easy way to obtain such an MXC URI is to upload an image to an (unencrypted) room and then copy the "url" from the source of the event.)
- `idp_brand`: An optional brand for this identity provider, allowing clients to style the login flow according to the identity provider in question. See the [spec](#) for possible options here.

- **discover**: set to false to disable the use of the OIDC discovery mechanism to discover endpoints. Defaults to true.
- **issuer**: Required. The OIDC issuer. Used to validate tokens and (if discovery is enabled) to discover the provider's endpoints.
- **client_id**: Required. oauth2 client id to use.
- **client_secret**: oauth2 client secret to use. May be omitted if **client_secret_jwt_key** is given, or if **client_auth_method** is 'none'. Must be omitted if **client_secret_path** is specified.
- **client_secret_path**: path to the oauth2 client secret to use. With that it's not necessary to leak secrets into the config file itself. Mutually exclusive with **client_secret**. Can be omitted if **client_secret_jwt_key** is specified.

Added in Synapse 1.91.0.

- **client_secret_jwt_key**: Alternative to **client_secret**: details of a key used to create a JSON Web Token to be used as an OAuth2 client secret. If given, must be a dictionary with the following properties:
 - **key**: a pem-encoded signing key. Must be a suitable key for the algorithm specified. Required unless **key_file** is given.
 - **key_file**: the path to file containing a pem-encoded signing key file. Required unless **key** is given.
 - **jwt_header**: a dictionary giving properties to include in the JWT header. Must include the key **alg**, giving the algorithm used to sign the JWT, such as "ES256", using the JWA identifiers in RFC7518.
 - **jwt_payload**: an optional dictionary giving properties to include in the JWT payload. Normally this should include an **iss** key.
- **client_auth_method**: auth method to use when exchanging the token. Valid values are **client_secret_basic** (default), **client_secret_post** and **none**.
- **pkce_method**: Whether to use proof key for code exchange when requesting and exchanging the token. Valid values are: **auto**, **always**, or **never**. Defaults to **auto**, which uses PKCE if supported during metadata discovery. Set to **always** to force enable PKCE or **never** to force disable PKCE.
- **scopes**: list of scopes to request. This should normally include the "openid" scope. Defaults to `["openid"]`.
- **authorization_endpoint**: the oauth2 authorization endpoint. Required if provider discovery is disabled.

- **token_endpoint**: the oauth2 token endpoint. Required if provider discovery is disabled.
- **userinfo_endpoint**: the OIDC userinfo endpoint. Required if discovery is disabled and the 'openid' scope is not requested.
- **jwks_uri**: URI where to fetch the JWKS. Required if discovery is disabled and the 'openid' scope is used.
- **skip_verification**: set to 'true' to skip metadata verification. Use this if you are connecting to a provider that is not OpenID Connect compliant. Defaults to false. Avoid this in production.
- **user_profile_method**: Whether to fetch the user profile from the userinfo endpoint, or to rely on the data returned in the id_token from the **token_endpoint**. Valid values are: **auto** or **userinfo_endpoint**. Defaults to **auto**, which uses the userinfo endpoint if **openid** is not included in **scopes**. Set to **userinfo_endpoint** to always use the userinfo endpoint.
- **additional_authorization_parameters**: String to string dictionary that will be passed as additional parameters to the authorization grant URL.
- **allow_existing_users**: set to true to allow a user logging in via OIDC to match a pre-existing account instead of failing. This could be used if switching from password logins to OIDC. Defaults to false.
- **enable_registration**: set to 'false' to disable automatic registration of new users. This allows the OIDC SSO flow to be limited to sign in only, rather than automatically registering users that have a valid SSO login but do not have a pre-registered account. Defaults to true.
- **user_mapping_provider**: Configuration for how attributes returned from a OIDC provider are mapped onto a matrix user. This setting has the following sub-properties:
 - **module**: The class name of a custom mapping module. Default is **synapse.handlers.oidc.JinjaOidcMappingProvider**. See [OpenID Mapping Providers](#) for information on implementing a custom mapping provider.
 - **config**: Configuration for the mapping provider module. This section will be passed as a Python dictionary to the user mapping provider module's **parse_config** method.

For the default provider, the following settings are available:

- **subject_template**: Jinja2 template for a unique identifier for the user. Defaults to `{{ user.sub }}`, which OpenID Connect compliant providers should provide.

This replaces and overrides `subject_claim`.

- `subject_claim`: name of the claim containing a unique identifier for the user. Defaults to 'sub', which OpenID Connect compliant providers should provide.

Deprecated in Synapse v1.75.0.

- `picture_template`: Jinja2 template for an url for the user's profile picture. Defaults to `{{ user.picture }}`, which OpenID Connect compliant providers should provide and has to refer to a direct image file such as PNG, JPEG, or GIF image file.

This replaces and overrides `picture_claim`.

Currently only supported in monolithic (single-process) server configurations where the media repository runs within the Synapse process.

- `picture_claim`: name of the claim containing an url for the user's profile picture. Defaults to 'picture', which OpenID Connect compliant providers should provide and has to refer to a direct image file such as PNG, JPEG, or GIF image file.

Currently only supported in monolithic (single-process) server configurations where the media repository runs within the Synapse process.

Deprecated in Synapse v1.75.0.

- `localpart_template`: Jinja2 template for the localpart of the MXID. If this is not set, the user will be prompted to choose their own username (see the documentation for the `sso_auth_account_details.html` template). This template can use the `localpart_from_email` filter.
- `confirm_localpart`: Whether to prompt the user to validate (or change) the generated localpart (see the documentation for the 'sso_auth_account_details.html' template), instead of registering the account right away.
- `display_name_template`: Jinja2 template for the display name to set on first login. If unset, no displayname will be set.
- `email_template`: Jinja2 template for the email address of the user. If unset, no email address will be added to the account.
- `extra_attributes`: a map of Jinja2 templates for extra attributes to send back to the client during login. Note that these are non-standard and clients will ignore them without modifications.

When rendering, the Jinja2 templates are given a 'user' variable, which is set to the claims returned by the UserInfo Endpoint and/or in the ID Token.

- `backchannel_logout_enabled`: set to `true` to process OIDC Back-Channel Logout notifications. Those notifications are expected to be received on `/_synapse/client/oidc/backchannel_logout`. Defaults to `false`.
- `backchannel_logout_ignore_sub`: by default, the OIDC Back-Channel Logout feature checks that the `sub` claim matches the subject claim received during login. This check can be disabled by setting this to `true`. Defaults to `false`.

You might want to disable this if the `subject_claim` returned by the mapping provider is not `sub`.

It is possible to configure Synapse to only allow logins if certain attributes match particular values in the OIDC userinfo. The requirements can be listed under `attribute_requirements` as shown here:

```
attribute_requirements:  
  - attribute: family_name  
    value: "Stephenson"  
  - attribute: groups  
    value: "admin"
```

All of the listed attributes must match for the login to be permitted. Additional attributes can be added to userinfo by expanding the `scopes` section of the OIDC config to retrieve additional information from the OIDC provider.

If the OIDC claim is a list, then the attribute must match any value in the list. Otherwise, it must exactly match the value of the claim. Using the example above, the `family_name` claim MUST be "Stephenson", but the `groups` claim MUST contain "admin".

Example configuration:

```

oidc_providers:
  # Generic example
  #
  - idp_id: my_idp
    idp_name: "My OpenID provider"
    idp_icon: "mxc://example.com/mediaid"
    discover: false
    issuer: "https://accounts.example.com/"
    client_id: "provided-by-your-issuer"
    client_secret: "provided-by-your-issuer"
    client_auth_method: client_secret_post
    scopes: ["openid", "profile"]
    authorization_endpoint: "https://accounts.example.com/oauth2/auth"
    token_endpoint: "https://accounts.example.com/oauth2/token"
    userinfo_endpoint: "https://accounts.example.com/userinfo"
    jwks_uri: "https://accounts.example.com/.well-known/jwks.json"
    additional_authorization_parameters:
      acr_values: 2fa
      skip_verification: true
      enable_registration: true
      user_mapping_provider:
        config:
          subject_claim: "id"
          localpart_template: "{{ user.login }}"
          display_name_template: "{{ user.name }}"
          email_template: "{{ user.email }}"
        attribute_requirements:
          - attribute: userGroup
            value: "synapseUsers"

```

cas_config

Enable Central Authentication Service (CAS) for registration and login. Has the following sub-options:

- **enabled**: Set this to true to enable authorization against a CAS server. Defaults to false.
- **idp_name**: A user-facing name for this identity provider, which is used to offer the user a choice of login mechanisms.
- **idp_icon**: An optional icon for this identity provider, which is presented by clients and Synapse's own IdP picker page. If given, must be an MXC URI of the format `mxc://<server-name>/<media-id>`. (An easy way to obtain such an MXC URI is to upload an image to an (unencrypted) room and then copy the "url" from the source of the event.)
- **idp_brand**: An optional brand for this identity provider, allowing clients to style the login flow according to the identity provider in question. See the [spec](#) for possible options here.

- **server_url**: The URL of the CAS authorization endpoint.
- **protocol_version**: The CAS protocol version, defaults to none (version 3 is required if you want to use "required_attributes").
- **displayname_attribute**: The attribute of the CAS response to use as the display name. If no name is given here, no displayname will be set.
- **required_attributes**: It is possible to configure Synapse to only allow logins if CAS attributes match particular values. All of the keys given below must exist and the values must match the given value. Alternately if the given value is **None** then any value is allowed (the attribute just must exist). All of the listed attributes must match for the login to be permitted.
- **enable_registration**: set to 'false' to disable automatic registration of new users. This allows the CAS SSO flow to be limited to sign in only, rather than automatically registering users that have a valid SSO login but do not have a pre-registered account. Defaults to true.
- **allow_numeric_ids**: set to 'true' allow numeric user IDs (default false). This allows CAS SSO flow to provide user IDs composed of numbers only. These identifiers will be prefixed by the letter "u" by default. The prefix can be configured using the "numeric_ids_prefix" option. Be careful to choose the prefix correctly to avoid any possible conflicts (e.g. user 1234 becomes u1234 when a user u1234 already exists).
- **numeric_ids_prefix**: the prefix you wish to add in front of a numeric user ID when the "allow_numeric_ids" option is set to "true". By default, the prefix is the letter "u" and only alphanumeric characters are allowed.

Added in Synapse 1.93.0.

Example configuration:

```
cas_config:
  enabled: true
  server_url: "https://cas-server.com"
  protocol_version: 3
  displayname_attribute: name
  required_attributes:
    userGroup: "staff"
    department: None
  enable_registration: true
  allow_numeric_ids: true
  numeric_ids_prefix: "numericuser"
```

SSO

Additional settings to use with single-sign on systems such as OpenID Connect, SAML2 and CAS.

Server admins can configure custom templates for pages related to SSO. See [here](#) for more information.

Options include:

- **client_whitelist**: A list of client URLs which are whitelisted so that the user does not have to confirm giving access to their account to the URL. Any client whose URL starts with an entry in the following list will not be subject to an additional confirmation step after the SSO login is completed. **WARNING:** An entry such as "https://my.client" is insecure, because it will also match "https://my.client.evil.site", exposing your users to phishing attacks from evil.site. To avoid this, include a slash after the hostname: "https://my.client/". The login fallback page (used by clients that don't natively support the required login flows) is whitelisted in addition to any URLs in this list. By default, this list contains only the login fallback page.
- **update_profile_information**: Use this setting to keep a user's profile fields in sync with information from the identity provider. Currently only syncing the displayname is supported. Fields are checked on every SSO login, and are updated if necessary. Note that enabling this option will override user profile information, regardless of whether users have opted-out of syncing that information when first signing in. Defaults to false.

Example configuration:

```
sso:  
  client_whitelist:  
    - https://riot.im/develop  
    - https://my.custom.client/  
  update_profile_information: true
```

jwt_config

JSON web token integration. The following settings can be used to make Synapse JSON web tokens for authentication, instead of its internal password database.

Each JSON Web Token needs to contain a "sub" (subject) claim, which is used as the localpart of the mxid.

Additionally, the expiration time ("exp"), not before time ("nbf"), and issued at ("iat") claims are validated if present.

Note that this is a non-standard login type and client support is expected to be non-existent.

See [here](#) for more.

Additional sub-options for this setting include:

- **enabled**: Set to true to enable authorization using JSON web tokens. Defaults to false.
- **secret**: This is either the private shared secret or the public key used to decode the contents of the JSON web token. Required if **enabled** is set to true.
- **algorithm**: The algorithm used to sign (or HMAC) the JSON web token. Supported algorithms are listed [here \(section JWS\)](#). Required if **enabled** is set to true.
- **subject_claim**: Name of the claim containing a unique identifier for the user. Optional, defaults to **sub**.
- **display_name_claim**: Name of the claim containing the display name for the user. Optional. If provided, the display name will be set to the value of this claim upon first login.
- **issuer**: The issuer to validate the "iss" claim against. Optional. If provided the "iss" claim will be required and validated for all JSON web tokens.
- **audiences**: A list of audiences to validate the "aud" claim against. Optional. If provided the "aud" claim will be required and validated for all JSON web tokens. Note that if the "aud" claim is included in a JSON web token then validation will fail without configuring audiences.

Example configuration:

```
jwt_config:
  enabled: true
  secret: "provided-by-your-issuer"
  algorithm: "provided-by-your-issuer"
  subject_claim: "name_of_claim"
  display_name_claim: "name_of_claim"
  issuer: "provided-by-your-issuer"
  audiences:
    - "provided-by-your-issuer"
```

password_config

Use this setting to enable password-based logins.

This setting has the following sub-options:

- **enabled**: Defaults to true. Set to false to disable password authentication. Set to **only_for_reauth** to allow users with existing passwords to use them to reauthenticate (not log in), whilst preventing new users from setting passwords.
- **localdb_enabled**: Set to false to disable authentication against the local password database. This is ignored if **enabled** is false, and is only useful if you have other **password_providers**. Defaults to true.

- **pepper**: Set the value here to a secret random string for extra security. DO NOT CHANGE THIS AFTER INITIAL SETUP!
- **policy**: Define and enforce a password policy, such as minimum lengths for passwords, etc. Each parameter is optional. This is an implementation of MSC2000. Parameters are as follows:
 - **enabled**: Defaults to false. Set to true to enable.
 - **minimum_length**: Minimum accepted length for a password. Defaults to 0.
 - **require_digit**: Whether a password must contain at least one digit. Defaults to false.
 - **require_symbol**: Whether a password must contain at least one symbol. A symbol is any character that's not a number or a letter. Defaults to false.
 - **require_lowercase**: Whether a password must contain at least one lowercase letter. Defaults to false.
 - **require_uppercase**: Whether a password must contain at least one uppercase letter. Defaults to false.

Example configuration:

```
password_config:
  enabled: false
  localdb_enabled: false
  pepper: "EVEN_MORE_SECRET"

policy:
  enabled: true
  minimum_length: 15
  require_digit: true
  require_symbol: true
  require_lowercase: true
  require_uppercase: true
```

Push

Configuration settings related to push notifications

push

This setting defines options for push notifications.

This option has a number of sub-options. They are as follows:

- **enabled**: Enables or disables push notification calculation. Note, disabling this will also stop unread counts being calculated for rooms. This mode of operation is intended for

homeservers which may only have bots or appservice users connected, or are otherwise not interested in push/unread counters. This is enabled by default.

- **include_content**: Clients requesting push notifications can either have the body of the message sent in the notification poke along with other details like the sender, or just the event ID and room ID (**event_id_only**). If clients choose the to have the body sent, this option controls whether the notification request includes the content of the event (other details like the sender are still included). If **event_id_only** is enabled, it has no effect. For modern android devices the notification content will still appear because it is loaded by the app. iPhone, however will send a notification saying only that a message arrived and who it came from. Defaults to true. Set to false to only include the event ID and room ID in push notification payloads.
- **group_unread_count_by_room: false**: When a push notification is received, an unread count is also sent. This number can either be calculated as the number of unread messages for the user, or the number of *rooms* the user has unread messages in. Defaults to true, meaning push clients will see the number of rooms with unread messages in them. Set to false to instead send the number of unread messages.
- **jitter_delay**: Delays push notifications by a random amount up to the given duration. Useful for mitigating timing attacks. Optional, defaults to no delay. *Added in Synapse 1.84.0.*

Example configuration:

```
push:
  enabled: true
  include_content: false
  group_unread_count_by_room: false
  jitter_delay: "10s"
```

Rooms

Config options relating to rooms.

encryption_enabled_by_default_for_room_type

Controls whether locally-created rooms should be end-to-end encrypted by default.

Possible options are "all", "invite", and "off". They are defined as:

- "all": any locally-created room
- "invite": any room created with the **private_chat** or **trusted_private_chat** room creation presets
- "off": this option will take no effect

The default value is "off".

Note that this option will only affect rooms created after it is set. It will also not affect rooms created by other servers.

Example configuration:

```
encryption_enabled_by_default_for_room_type: invite
```

user_directory

This setting defines options related to the user directory.

This option has the following sub-options:

- **enabled**: Defines whether users can search the user directory. If false then empty responses are returned to all queries. Defaults to true.
- **search_all_users**: Defines whether to search all users visible to your homeserver at the time the search is performed. If set to true, will return all users known to the homeserver matching the search query. If false, search results will only contain users visible in public rooms and users sharing a room with the requester. Defaults to false.

NB. If you set this to true, and the last time the user_directory search indexes were (re)built was before Synapse 1.44, you'll have to rebuild the indexes in order to search through all known users.

These indexes are built the first time Synapse starts; admins can manually trigger a rebuild via the API following the instructions [for running background updates](#), set to true to return search results containing all known users, even if that user does not share a room with the requester.

- **prefer_local_users**: Defines whether to prefer local users in search query results. If set to true, local users are more likely to appear above remote users when searching the user directory. Defaults to false.
- **show_locked_users**: Defines whether to show locked users in search query results. Defaults to false.

Example configuration:

```
user_directory:  
  enabled: false  
  search_all_users: true  
  prefer_local_users: true  
  show_locked_users: true
```

user_consent

For detailed instructions on user consent configuration, see [here](#).

Parts of this section are required if enabling the `consent` resource under `listeners`, in particular `template_dir` and `version`.

- `template_dir`: gives the location of the templates for the HTML forms. This directory should contain one subdirectory per language (eg, `en`, `fr`), and each language directory should contain the policy document (named as `.html`) and a success page (`success.html`).
- `version`: specifies the 'current' version of the policy document. It defines the version to be served by the consent resource if there is no 'v' parameter.
- `server_notice_content`: if enabled, will send a user a "Server Notice" asking them to consent to the privacy policy. The `server_notices` section must also be configured for this to work. Notices will *not* be sent to guest users unless `send_server_notice_to_guests` is set to true.
- `block_events_error`, if set, will block any attempts to send events until the user consents to the privacy policy. The value of the setting is used as the text of the error.
- `require_at_registration`, if enabled, will add a step to the registration process, similar to how captcha works. Users will be required to accept the policy before their account is created.
- `policy_name` is the display name of the policy users will see when registering for an account. Has no effect unless `require_at_registration` is enabled. Defaults to "Privacy Policy".

Example configuration:

```
user_consent:
  template_dir: res/templates/privacy
  version: 1.0
  server_notice_content:
    msgtype: m.text
    body: >-
      To continue using this homeserver you must review and agree to the
      terms and conditions at %(consent_uris)
  send_server_notice_to_guests: true
  block_events_error: >-
    To continue using this homeserver you must review and agree to the
    terms and conditions at %(consent_uris)
  require_at_registration: false
  policy_name: Privacy Policy
```

stats

Settings for local room and user statistics collection. See [here](#) for more.

- **enabled**: Set to false to disable room and user statistics. Note that doing so may cause certain features (such as the room directory) not to work correctly. Defaults to true.

Example configuration:

```
stats:  
  enabled: false
```

server_notices

Use this setting to enable a room which can be used to send notices from the server to users. It is a special room which users cannot leave; notices in the room come from a special "notices" user id.

If you use this setting, you *must* define the **system_mxid_localpart** sub-setting, which defines the id of the user which will be used to send the notices.

Sub-options for this setting include:

- **system_mxid_display_name**: set the display name of the "notices" user
- **system_mxid_avatar_url**: set the avatar for the "notices" user
- **room_name**: set the room name of the server notices room
- **room_avatar_url**: optional string. The room avatar to use for server notice rooms. If set to the empty string "", notice rooms will not be given an avatar. Defaults to the empty string. *Added in Synapse 1.99.0.*
- **room_topic**: optional string. The topic to use for server notice rooms. If set to the empty string "", notice rooms will not be given a topic. Defaults to the empty string. *Added in Synapse 1.99.0.*
- **auto_join**: boolean. If true, the user will be automatically joined to the room instead of being invited. Defaults to false. *Added in Synapse 1.98.0.*

Note that the name, topic and avatar of existing server notice rooms will only be updated when a new notice event is sent.

Example configuration:

```
server_notices:
  system_mxid_localpart: notices
  system_mxid_display_name: "Server Notices"
  system_mxid_avatar_url: "mxc://example.com/oumMVlgDnLYFaPVkExemNVVZ"
  room_name: "Server Notices"
  room_avatar_url: "mxc://example.com/oumMVlgDnLYFaPVkExemNVVZ"
  room_topic: "Room used by your server admin to notice you of important
  information"
  auto_join: true
```

enable_room_list_search

Set to false to disable searching the public room list. When disabled blocks searching local and remote room lists for local and remote users by always returning an empty list for all queries. Defaults to true.

Example configuration:

```
enable_room_list_search: false
```

alias_creation_rules

The `alias_creation_rules` option allows server admins to prevent unwanted alias creation on this server.

This setting is an optional list of 0 or more rules. By default, no list is provided, meaning that all alias creations are permitted.

Otherwise, requests to create aliases are matched against each rule in order. The first rule that matches decides if the request is allowed or denied. If no rule matches, the request is denied. In particular, this means that configuring an empty list of rules will deny every alias creation request.

Each rule is a YAML object containing four fields, each of which is an optional string:

- `user_id`: a glob pattern that matches against the creator of the alias.
- `alias`: a glob pattern that matches against the alias being created.
- `room_id`: a glob pattern that matches against the room ID the alias is being pointed at.
- `action`: either `allow` or `deny`. What to do with the request if the rule matches.
Defaults to `allow`.

Each of the glob patterns is optional, defaulting to `*` ("match anything"). Note that the patterns match against fully qualified IDs, e.g. against `@alice@example.com`,

#room:example.com and !abcdefgijk:example.com instead of alice, room and abcedgghijk.

Example configuration:

```
# No rule list specified. All alias creations are allowed.
# This is the default behaviour.
alias_creation_rules:
```

```
# A list of one rule which allows everything.
# This has the same effect as the previous example.
alias_creation_rules:
  - "action": "allow"
```

```
# An empty list of rules. All alias creations are denied.
alias_creation_rules: []
```

```
# A list of one rule which denies everything.
# This has the same effect as the previous example.
alias_creation_rules:
  - "action": "deny"
```

```
# Prevent a specific user from creating aliases.
# Allow other users to create any alias
alias_creation_rules:
  - user_id: "@bad_user@example.com"
    action: deny

  - action: allow
```

```
# Prevent aliases being created which point to a specific room.
alias_creation_rules:
  - room_id: "!forbiddenRoom@example.com"
    action: deny

  - action: allow
```

room_list_publication_rules

The `room_list_publication_rules` option allows server admins to prevent unwanted entries from being published in the public room list.

The format of this option is the same as that for `alias_creation_rules`: an optional list of 0 or more rules. By default, no list is provided, meaning that all rooms may be published to the room list.

Otherwise, requests to publish a room are matched against each rule in order. The first rule that matches decides if the request is allowed or denied. If no rule matches, the request is

denied. In particular, this means that configuring an empty list of rules will deny every alias creation request.

Requests to create a public (public as in published to the room directory) room which violates the configured rules will result in the room being created but not published to the room directory.

Each rule is a YAML object containing four fields, each of which is an optional string:

- `user_id`: a glob pattern that matches against the user publishing the room.
- `alias`: a glob pattern that matches against one of published room's aliases.
 - If the room has no aliases, the alias match fails unless `alias` is unspecified or `*`.
 - If the room has exactly one alias, the alias match succeeds if the `alias` pattern matches that alias.
 - If the room has two or more aliases, the alias match succeeds if the pattern matches at least one of the aliases.
- `room_id`: a glob pattern that matches against the room ID of the room being published.
- `action`: either `allow` or `deny`. What to do with the request if the rule matches. Defaults to `allow`.

Each of the glob patterns is optional, defaulting to `*` ("match anything"). Note that the patterns match against fully qualified IDs, e.g. against `@alice@example.com`, `#room@example.com` and `!abcdefgijk@example.com` instead of `alice`, `room` and `abcdefgijk`.

Example configuration:

```
# No rule list specified. Anyone may publish any room to the public list.
# This is the default behaviour.
room_list_publication_rules:
```

```
# A list of one rule which allows everything.
# This has the same effect as the previous example.
room_list_publication_rules:
  - "action": "allow"
```

```
# An empty list of rules. No-one may publish to the room list.
room_list_publication_rules: []
```

```
# A list of one rule which denies everything.
# This has the same effect as the previous example.
room_list_publication_rules:
  - "action": "deny"
```

```
# Prevent a specific user from publishing rooms.
# Allow other users to publish anything.
room_list_publication_rules:
  - user_id: "@bad_user@example.com"
    action: deny

  - action: allow
```

```
# Prevent publication of a specific room.
room_list_publication_rules:
  - room_id: "!forbiddenRoom@example.com"
    action: deny

  - action: allow
```

```
# Prevent publication of rooms with at least one alias containing the word
"potato".
room_list_publication_rules:
  - alias: "#*potato*:example.com"
    action: deny

  - action: allow
```

default_power_level_content_override

The `default_power_level_content_override` option controls the default power levels for rooms.

Useful if you know that your users need special permissions in rooms that they create (e.g. to send particular types of state events without needing an elevated power level). This takes the same shape as the `power_level_content_override` parameter in the /createRoom API, but is applied before that parameter.

Note that each key provided inside a preset (for example `events` in the example below) will overwrite all existing defaults inside that key. So in the example below, newly-created private_chat rooms will have no rules for any event types except `com.example.foo`.

Example configuration:

```
default_power_level_content_override:
  private_chat: { "events": { "com.example.foo" : 0 } }
  trusted_private_chat: null
  public_chat: null
```

The default power levels for each preset are:

```
"m.room.name": 50
"m.room.power_levels": 100
"m.room.history_visibility": 100
"m.room.canonical_alias": 50
"m.room.avatar": 50
"m.room.tombstone": 100
"m.room.server_acl": 100
"m.room.encryption": 100
```

So a complete example where the default power-levels for a preset are maintained but the power level for a new key is set is:

```
default_power_level_content_override:
  private_chat:
    events:
      "com.example.foo": 0
      "m.room.name": 50
      "m.room.power_levels": 100
      "m.room.history_visibility": 100
      "m.room.canonical_alias": 50
      "m.room.avatar": 50
      "m.room.tombstone": 100
      "m.room.server_acl": 100
      "m.room.encryption": 100
    trusted_private_chat: null
  public_chat: null
```

forget_rooms_on_leave

Set to true to automatically forget rooms for users when they leave them, either normally or via a kick or ban. Defaults to false.

Example configuration:

```
forget_rooms_on_leave: false
```

exclude_rooms_from_sync

A list of rooms to exclude from sync responses. This is useful for server administrators wishing to group users into a room without these users being able to see it from their client.

By default, no room is excluded.

Example configuration:

```
exclude_rooms_from_sync:
  - "!foo@example.com"
```

Opentracing

Configuration options related to Opentracing support.

opentracing

These settings enable and configure opentracing, which implements distributed tracing. This allows you to observe the causal chains of events across servers including requests, key lookups etc., across any server running synapse or any other services which support opentracing (specifically those implemented with Jaeger).

Sub-options include:

- `enabled`: whether tracing is enabled. Set to true to enable. Disabled by default.
- `homeserver_whitelist`: The list of homeservers we wish to send and receive span contexts and span baggage. See [here](#) for more. This is a list of regexes which are matched against the `server_name` of the homeserver. By default, it is empty, so no servers are matched.
- `force_tracing_for_users`: # A list of the matrix IDs of users whose requests will always be traced, even if the tracing system would otherwise drop the traces due to probabilistic sampling. By default, the list is empty.
- `jaeger_config`: Jaeger can be configured to sample traces at different rates. All configuration options provided by Jaeger can be set here. Jaeger's configuration is mostly related to trace sampling which is documented [here](#).

Example configuration:

```
opentracing:  
  enabled: true  
  homeserver_whitelist:  
    - ".*"  
  force_tracing_for_users:  
    - "@user1:server_name"  
    - "@user2:server_name"  
  
  jaeger_config:  
    sampler:  
      type: const  
      param: 1  
    logging:  
      false
```

Coordinating workers

Configuration options related to workers which belong in the main config file (usually called `homeserver.yaml`). A Synapse deployment can scale horizontally by running multiple Synapse processes called *workers*. Incoming requests are distributed between workers to handle higher loads. Some workers are privileged and can accept requests from other workers.

As a result, the worker configuration is divided into two parts.

1. The first part (in this section of the manual) defines which shardable tasks are delegated to privileged workers. This allows unprivileged workers to make requests to a privileged worker to act on their behalf.
2. [The second part](#) controls the behaviour of individual workers in isolation.

For guidance on setting up workers, see the [worker documentation](#).

`worker_replication_secret`

A shared secret used by the replication APIs on the main process to authenticate HTTP requests from workers.

The default, this value is omitted (equivalently `null`), which means that traffic between the workers and the main process is not authenticated.

Replacing an existing `worker_replication_secret` with a new one will break communication with all workers that have not yet updated their secret.

Example configuration:

```
worker_replication_secret: "secret_secret"
```

`start_pushers`

Unnecessary to set if using `pusher_instances` with `generic_workers`.

Controls sending of push notifications on the main process. Set to `false` if using a [pusher worker](#). Defaults to `true`.

Example configuration:

```
start_pushers: false
```

pusher_instances

It is possible to scale the processes that handle sending push notifications to [sygnal](#) and email by running a `generic_worker` and adding it's `worker_name` to a `pusher_instances` map. Doing so will remove handling of this function from the main process. Multiple workers can be added to this map, in which case the work is balanced across them. Ensure the main process and all pusher workers are restarted after changing this option.

Example configuration for a single worker:

```
pusher_instances:  
  - pusher_worker1
```

And for multiple workers:

```
pusher_instances:  
  - pusher_worker1  
  - pusher_worker2
```

send_federation

Unnecessary to set if using `federation_sender_instances` with `generic_workers`.

Controls sending of outbound federation transactions on the main process. Set to `false` if using a [federation sender worker](#). Defaults to `true`.

Example configuration:

```
send_federation: false
```

federation_sender_instances

It is possible to scale the processes that handle sending outbound federation requests by running a `generic_worker` and adding it's `worker_name` to a `federation_sender_instances` map. Doing so will remove handling of this function from the main process. Multiple workers can be added to this map, in which case the work is balanced across them.

The way that the load balancing works is any outbound federation request will be assigned to a federation sender worker based on the hash of the destination server name. This means that all requests being sent to the same destination will be processed by the same worker instance. Multiple `federation_sender_instances` are useful if there is a federation with multiple servers.

This configuration setting must be shared between all workers handling federation sending, and if changed all federation sender workers must be stopped at the same time and then started, to ensure that all instances are running with the same config (otherwise events may be dropped).

Example configuration for a single worker:

```
federation_sender_instances:
  - federation_sender1
```

And for multiple workers:

```
federation_sender_instances:
  - federation_sender1
  - federation_sender2
```

instance_map

When using workers this should be a map from `worker_name` to the HTTP replication listener of the worker, if configured, and to the main process. Each worker declared under `stream_writers` and `outbound_federation_restricted_to` needs a HTTP replication listener, and that listener should be included in the `instance_map`. The main process also needs an entry on the `instance_map`, and it should be listed under `main` **if even one other worker exists**. Ensure the port matches with what is declared inside the `listener` block for a `replication` listener.

Example configuration:

```
instance_map:
  main:
    host: localhost
    port: 8030
  worker1:
    host: localhost
    port: 8034
  other:
    host: localhost
    port: 8035
    tls: true
```

Example configuration(#2, for UNIX sockets):

```
instance_map:
  main:
    path: /run/synapse/main_replication.sock
  worker1:
    path: /run/synapse/worker1_replication.sock
```

stream_writers

Experimental: When using workers you can define which workers should handle writing to streams such as event persistence and typing notifications. Any worker specified here must also be in the `instance_map`.

See the list of available streams in the [worker documentation](#).

Example configuration:

```
stream_writers:  
  events: worker1  
  typing: worker1
```

outbound_federation_restricted_to

When using workers, you can restrict outbound federation traffic to only go through a specific subset of workers. Any worker specified here must also be in the `instance_map`. `worker_replication_secret` must also be configured to authorize inter-worker communication.

```
outbound_federation_restricted_to:  
  - federation_sender1  
  - federation_sender2
```

Also see the [worker documentation](#) for more info.

Added in Synapse 1.89.0.

run_background_tasks_on

The [worker](#) that is used to run background tasks (e.g. cleaning up expired data). If not provided this defaults to the main process.

Example configuration:

```
run_background_tasks_on: worker1
```

update_user_directory_from_worker

The [worker](#) that is used to update the user directory. If not provided this defaults to the main process.

Example configuration:

```
update_user_directory_from_worker: worker1
```

Added in Synapse 1.59.0.

notify_appservices_from_worker

The [worker](#) that is used to send output traffic to Application Services. If not provided this defaults to the main process.

Example configuration:

```
notify_appservices_from_worker: worker1
```

Added in Synapse 1.59.0.

media_instance_running_background_jobs

The [worker](#) that is used to run background tasks for media repository. If running multiple media repositories you must configure a single instance to run the background tasks. If not provided this defaults to the main process or your single [media_repository](#) worker.

Example configuration:

```
media_instance_running_background_jobs: worker1
```

Added in Synapse 1.16.0.

redis

Configuration for Redis when using workers. This *must* be enabled when using workers. This setting has the following sub-options:

- `enabled`: whether to use Redis support. Defaults to false.
- `host` and `port`: Optional host and port to use to connect to redis. Defaults to localhost and 6379
- `path`: The full path to a local Unix socket file. **If this is used, `host` and `port` are ignored.** Defaults to `/tmp/redis.sock`

- **password**: Optional password if configured on the Redis instance.
- **password_path**: Alternative to **password**, reading the password from an external file. The file should be a plain text file, containing only the password. Synapse reads the password from the given file once at startup.
- **dbid**: Optional redis dbid if needs to connect to specific redis logical db.
- **use_tls**: Whether to use tls connection. Defaults to false.
- **certificate_file**: Optional path to the certificate file
- **private_key_file**: Optional path to the private key file
- **ca_file**: Optional path to the CA certificate file. Use this one or:
- **ca_path**: Optional path to the folder containing the CA certificate file

Added in Synapse 1.78.0.

Changed in Synapse 1.84.0: Added use_tls, certificate_file, private_key_file, ca_file and ca_path attributes

Changed in Synapse 1.85.0: Added path option to use a local Unix socket

Changed in Synapse 1.116.0: Added password_path

Example configuration:

```
redis:
  enabled: true
  host: localhost
  port: 6379
  password_path: <path_to_the_password_file>
  # OR password: <secret_password>
  dbid: <dbid>
  #use_tls: True
  #certificate_file: <path_to_the_certificate_file>
  #private_key_file: <path_to_the_private_key_file>
  #ca_file: <path_to_the_ca_certificate_file>
```

Individual worker configuration

These options configure an individual worker, in its worker configuration file. They should be not be provided when configuring the main process.

Note also the configuration above for [coordinating a cluster of workers](#).

For guidance on setting up workers, see the [worker documentation](#).

worker_app

The type of worker. The currently available worker applications are listed in [worker documentation](#).

The most common worker is the `synapse.app.generic_worker`.

Example configuration:

```
worker_app: synapse.app.generic_worker
```

worker_name

A unique name for the worker. The worker needs a name to be addressed in further parameters and identification in log files. We strongly recommend giving each worker a unique `worker_name`.

Example configuration:

```
worker_name: generic_worker1
```

worker_listeners

A worker can handle HTTP requests. To do so, a `worker_listeners` option must be declared, in the same way as the `listeners` option in the shared config.

Workers declared in `stream_writers` and `instance_map` will need to include a `replication` listener here, in order to accept internal HTTP requests from other workers.

Example configuration:

```
worker_listeners:
  - type: http
    port: 8083
    resources:
      - names: [client, federation]
```

Example configuration(#2, using UNIX sockets with a `replication` listener):

```
worker_listeners:
- type: http
  path: /run/synapse/worker_replication.sock
  resources:
    - names: [replication]
- type: http
  path: /run/synapse/worker_public.sock
  resources:
    - names: [client, federation]
```

worker_manhole

A worker may have a listener for `manhole`. It allows server administrators to access a Python shell on the worker.

Example configuration:

```
worker_manhole: 9000
```

This is a short form for:

```
worker_listeners:
- port: 9000
  bind_addresses: ['127.0.0.1']
  type: manhole
```

It needs also an additional `manhole_settings` configuration.

worker_daemonize

Specifies whether the worker should be started as a daemon process. If Synapse is being managed by `systemd`, this option must be omitted or set to `false`.

Defaults to `false`.

Example configuration:

```
worker_daemonize: true
```

worker_pid_file

When running a worker as a daemon, we need a place to store the `PID` of the worker. This option defines the location of that "pid file".

This option is required if `worker_daemonize` is `true` and ignored otherwise. It has no default.

See also the `pid_file` option for the main Synapse process.

Example configuration:

```
worker_pid_file: DATADIR/generic_worker1.pid
```

worker_log_config

This option specifies a yaml python logging config file as described [here](#). See also the `log_config` option for the main Synapse process.

Example configuration:

```
worker_log_config: /etc/matrix-synapse/generic-worker-log.yaml
```

Background Updates

Configuration settings related to background updates.

background_updates

Background updates are database updates that are run in the background in batches. The duration, minimum batch size, default batch size, whether to sleep between batches and if so, how long to sleep can all be configured. This is helpful to speed up or slow down the updates. This setting has the following sub-options:

- `background_update_duration_ms`: How long in milliseconds to run a batch of background updates for. Defaults to 100. Set a different time to change the default.
- `sleep_enabled`: Whether to sleep between updates. Defaults to true. Set to false to change the default.
- `sleep_duration_ms`: If sleeping between updates, how long in milliseconds to sleep for. Defaults to 1000. Set a duration to change the default.
- `min_batch_size`: Minimum size a batch of background updates can be. Must be greater than 0. Defaults to 1. Set a size to change the default.
- `default_batch_size`: The batch size to use for the first iteration of a new background update. The default is 100. Set a size to change the default.

Example configuration:

```
background_updates:
  background_update_duration_ms: 500
  sleep_enabled: false
  sleep_duration_ms: 300
  min_batch_size: 10
  default_batch_size: 50
```

Auto Accept Invites

Configuration settings related to automatically accepting invites.

auto_accept_invites

Automatically accepting invites controls whether users are presented with an invite request or if they are instead automatically joined to a room when receiving an invite. Set the `enabled` sub-option to true to enable auto-accepting invites. Defaults to false. This setting has the following sub-options:

- `enabled`: Whether to run the auto-accept invites logic. Defaults to false.
- `only_for_direct_messages`: Whether invites should be automatically accepted for all room types, or only for direct messages. Defaults to false.
- `only_from_local_users`: Whether to only automatically accept invites from users on this homeserver. Defaults to false.
- `worker_to_run_on`: Which worker to run this module on. This must match the "worker_name". If not set or `null`, invites will be accepted on the main process.

NOTE: Care should be taken not to enable this setting if the `synapse_auto_accept_invite` module is enabled and installed. The two modules will compete to perform the same task and may result in undesired behaviour. For example, multiple join events could be generated from a single invite.

Example configuration:

```
auto_accept_invites:
  enabled: true
  only_for_direct_messages: true
  only_from_local_users: true
  worker_to_run_on: "worker_1"
```

Homeserver Sample Configuration File

Below is a sample homeserver configuration file. The homeserver configuration file can be tweaked to change the behaviour of your homeserver. A restart of the server is generally required to apply any changes made to this file.

Note that the contents below are *not* intended to be copied and used as the basis for a real homeserver.yaml. Instead, if you are starting from scratch, please generate a fresh config using Synapse by following the instructions in [Installation](#).

Documentation for all configuration options can be found in the [Configuration Manual](#).

```
# This file is maintained as an up-to-date snapshot of the default
# homeserver.yaml configuration generated by Synapse. You can find a
# complete accounting of possible configuration options at
# https://element-
hq.github.io/synapse/latest/usage/configuration/config_documentation.html
#
# It is *not* intended to be copied and used as the basis for a real
# homeserver.yaml. Instead, if you are starting from scratch, please generate
# a fresh config using Synapse by following the instructions in
# https://element-hq.github.io/synapse/latest/setup/installation.html.
#
#####
# Configuration file for Synapse.
#
# This is a YAML file: see [1] for a quick introduction. Note in particular
# that *indentation is important*: all the elements of a list or dictionary
# should have the same indentation.
#
# [1]
https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html
#
# For more information on how to configure Synapse, including a complete
# accounting of
# each option, go to docs/usage/configuration/config_documentation.md or
# https://element-
hq.github.io/synapse/latest/usage/configuration/config_documentation.html
server_name: "SERVERNAME"
pid_file: DATADIR/homeserver.pid
listeners:
  - port: 8008
    tls: false
    type: http
    x_forwarded: true
    bind_addresses: ['::1', '127.0.0.1']
    resources:
      - names: [client, federation]
        compress: false
database:
  name: sqlite3
  args:
    database: DATADIR/homeserver.db
log_config: "CONFDIR/SERVERNAME.log.config"
media_store_path: DATADIR/media_store
signing_key_path: "CONFDIR/SERVERNAME.signing.key"
trusted_key_servers:
  - server_name: "matrix.org"
```

Logging Sample Configuration File

Below is a sample logging configuration file. This file can be tweaked to control how your homeserver will output logs. The value of the `log_config` option in your homeserver config should be the path to this file.

To apply changes made to this file, send Synapse a SIGHUP signal (or, if using `systemd`, run `systemctl reload` on the Synapse service).

Note that a default logging configuration (shown below) is created automatically alongside the homeserver config when following the [installation instructions](#). It should be named `<SERVERNAME>.log.config` by default.

Hint: If you're looking for a guide on what each of the fields in the "Processed request" log lines mean, see [Request log format](#).

```
# Log configuration for Synapse.
#
# This is a YAML file containing a standard Python logging configuration
# dictionary. See [1] for details on the valid settings.
#
# Synapse also supports structured logging for machine readable logs which can
# be ingested by ELK stacks. See [2] for details.
#
# [1]: https://docs.python.org/3/library/logging.config.html#configuration-dictionary-schema
# [2]: https://element-hq.github.io/synapse/latest/structured\_logging.html

version: 1

formatters:
    precise:
        format: '%(asctime)s - %(name)s - %(lineno)d - %(levelname)s - %
(request)s - %(message)s'

handlers:
    file:
        class: logging.handlers.TimedRotatingFileHandler
        formatter: precise
        filename: /var/log/matrix-synapse/homeserver.log
        when: midnight
        backupCount: 3 # Does not include the current log file.
        encoding: utf8

        # Default to buffering writes to log file for efficiency.
        # WARNING/ERROR logs will still be flushed immediately, but there will be a
        # delay (of up to `period` seconds, or until the buffer is full with
        # `capacity` messages) before INFO/DEBUG logs get written.

    buffer:
        class: synapse.logging.handlers.PeriodicallyFlushingMemoryHandler
        target: file

        # The capacity is the maximum number of log lines that are buffered
        # before being written to disk. Increasing this will lead to better
        # performance, at the expense of it taking longer for log lines to
        # be written to disk.
        # This parameter is required.
        capacity: 10

        # Logs with a level at or above the flush level will cause the buffer to
        # be flushed immediately.
        # Default value: 40 (ERROR)
        # Other values: 50 (CRITICAL), 30 (WARNING), 20 (INFO), 10 (DEBUG)
        flushLevel: 30 # Flush immediately for WARNING logs and higher

        # The period of time, in seconds, between forced flushes.
        # Messages will not be delayed for longer than this time.
        # Default value: 5 seconds
        period: 5

    console:
        # A handler that writes logs to stderr. Unused by default, but can be used
        # instead of "buffer" and "file" in the logger handlers.
```

```
class: logging.StreamHandler
formatter: precise

loggers:
  synapse.storage.SQL:
    # beware: increasing this to DEBUG will make synapse log sensitive
    # information such as access tokens.
    level: INFO

root:
  level: INFO

  # Write logs to the `buffer` handler, which will buffer them together in
  memory,
  # then write them to a file.
  #
  # Replace "buffer" with "console" to log to stderr instead.
  #
  handlers: [buffer]

disable_existing_loggers: false
```

Structured Logging

A structured logging system can be useful when your logs are destined for a machine to parse and process. By maintaining its machine-readable characteristics, it enables more efficient searching and aggregations when consumed by software such as the [ELK stack](#).

Synapse's structured logging system is configured via the file that Synapse's `log_config` config option points to. The file should include a formatter which uses the `synapse.logging.TerseJsonFormatter` class included with Synapse and a handler which uses the above formatter.

There is also a `synapse.logging.JsonFormatter` option which does not include a timestamp in the resulting JSON. This is useful if the log ingester adds its own timestamp.

A structured logging configuration looks similar to the following:

```
version: 1

formatters:
    structured:
        class: synapse.logging.TerseJsonFormatter

handlers:
    file:
        class: logging.handlers.TimedRotatingFileHandler
        formatter: structured
        filename: /path/to/my/logs/homeserver.log
        when: midnight
        backupCount: 3 # Does not include the current log file.
        encoding: utf8

loggers:
    synapse:
        level: INFO
        handlers: [remote]
    synapse.storage.SQL:
        level: WARNING
```

The above logging config will set Synapse as 'INFO' logging level by default, with the SQL layer at 'WARNING', and will log to a file, stored as JSON.

It is also possible to configure Synapse to log to a remote endpoint by using the `synapse.logging.RemoteHandler` class included with Synapse. It takes the following arguments:

- `host`: Hostname or IP address of the log aggregator.
- `port`: Numerical port to contact on the host.
- `maximum_buffer`: (Optional, defaults to 1000) The maximum buffer size to allow.

A remote structured logging configuration looks similar to the following:

```
version: 1

formatters:
  structured:
    class: synapse.logging.TerseJsonFormatter

handlers:
  remote:
    class: synapse.logging.RemoteHandler
    formatter: structured
    host: 10.1.2.3
    port: 9999

loggers:
  synapse:
    level: INFO
    handlers: [remote]
  synapse.storage.SQL:
    level: WARNING
```

The above logging config will set Synapse as 'INFO' logging level by default, with the SQL layer at 'WARNING', and will log JSON formatted messages to a remote endpoint at 10.1.2.3:9999.

Templates

Synapse uses parametrised templates to generate the content of emails it sends and webpages it shows to users.

By default, Synapse will use the templates listed [here](#). Server admins can configure an additional directory for Synapse to look for templates in, allowing them to specify custom templates:

```
templates:  
  custom_template_directory: /path/to/custom/templates/
```

If this setting is not set, or the files named below are not found within the directory, default templates from within the Synapse package will be used.

Templates that are given variables when being rendered are rendered using [Jinja 2](#). Templates rendered by Jinja 2 can also access two functions on top of the functions already available as part of Jinja 2:

```
format_ts(value: int, format: str) -> str
```

Formats a timestamp in milliseconds.

Example: `reason.last_sent_ts|format_ts("%c")`

```
mxc_to_http(value: str, width: int, height: int, resize_method: str = "crop") -> str
```

Turns a `mxc://` URL for media content into an HTTP(S) one using the homeserver's `public_baseurl` configuration setting as the URL's base.

Example: `message.sender_avatar_url|mxc_to_http(32,32)`

```
localpart_from_email(address: str) -> str
```

Returns the local part of an email address (e.g. `alice` in `alice@example.com`).

Example: `user.email_address|localpart_from_email`

Email templates

Below are the templates Synapse will look for when generating the content of an email:

- `notif_mail.html` and `notif_mail.txt`: The contents of email notifications of missed events. When rendering, this template is given the following variables:
 - `user_display_name`: the display name for the user receiving the notification
 - `unsubscribe_link`: the link users can click to unsubscribe from email notifications
 - `summary_text`: a summary of the notification(s). The text used can be customised by configuring the various settings in the `email.subjects` section of the configuration file.
 - `rooms`: a list of rooms containing events to include in the email. Each element is an object with the following attributes:
 - `title`: a human-readable name for the room
 - `hash`: a hash of the ID of the room
 - `invite`: a boolean, which is `True` if the room is an invite the user hasn't accepted yet, `False` otherwise
 - `notifs`: a list of events, or an empty list if `invite` is `True`. Each element is an object with the following attributes:
 - `link`: a `matrix.to` link to the event
 - `ts`: the time in milliseconds at which the event was received
 - `messages`: a list of messages containing one message before the event, the message in the event, and one message after the event. Each element is an object with the following attributes:
 - `event_type`: the type of the event
 - `is_historical`: a boolean, which is `False` if the message is the one that triggered the notification, `True` otherwise
 - `id`: the ID of the event
 - `ts`: the time in milliseconds at which the event was sent
 - `sender_name`: the display name for the event's sender
 - `sender_avatar_url`: the avatar URL (as a `mxc://` URL) for the event's sender
 - `sender_hash`: a hash of the user ID of the sender
 - `msgtype`: the type of the message
 - `body_text_html`: html representation of the message
 - `body_text_plain`: plaintext representation of the message
 - `image_url`: `mxc` url of an image, when "msgtype" is "m.image"
 - `link`: a `matrix.to` link to the room
 - `avator_url`: url to the room's avator
 - `reason`: information on the event that triggered the email to be sent. It's an object with the following attributes:
 - `room_id`: the ID of the room the event was sent in
 - `room_name`: a human-readable name for the room the event was sent in
 - `now`: the current time in milliseconds
 - `received_at`: the time in milliseconds at which the event was received

- `delay_before_mail_ms`: the amount of time in milliseconds Synapse always waits before ever emailing about a notification (to give the user a chance to respond to other push or notice the window)
- `last_sent_ts`: the time in milliseconds at which a notification was last sent for an event in this room
- `throttle_ms`: the minimum amount of time in milliseconds between two notifications can be sent for this room
- `password_reset.html` and `password_reset.txt`: The contents of password reset emails sent by the homeserver. When rendering, these templates are given a `link` variable which contains the link the user must click in order to reset their password.
- `registration.html` and `registration.txt`: The contents of address verification emails sent during registration. When rendering, these templates are given a `link` variable which contains the link the user must click in order to validate their email address.
- `add_threepid.html` and `add_threepid.txt`: The contents of address verification emails sent when an address is added to a Matrix account. When rendering, these templates are given a `link` variable which contains the link the user must click in order to validate their email address.

HTML page templates for registration and password reset

Below are the templates Synapse will look for when generating pages related to registration and password reset:

- `password_reset_confirmation.html`: An HTML page that a user will see when they follow the link in the password reset email. The user will be asked to confirm the action before their password is reset. When rendering, this template is given the following variables:
 - `sid`: the session ID for the password reset
 - `token`: the token for the password reset
 - `client_secret`: the client secret for the password reset
- `password_reset_success.html` and `password_reset_failure.html`: HTML pages for success and failure that a user will see when they confirm the password reset flow using the page above. When rendering, `password_reset_success.html` is given no variable, and `password_reset_failure.html` is given a `failure_reason`, which contains the reason for the password reset failure.
- `registration_success.html` and `registration_failure.html`: HTML pages for success and failure that a user will see when they follow the link in an address verification email sent during registration. When rendering, `registration_success.html` is given no variable, and `registration_failure.html` is given a `failure_reason`, which contains the reason for the registration failure.

- `add_threepid_success.html` and `add_threepid_failure.html`: HTML pages for success and failure that a user will see when they follow the link in an address verification email sent when an address is added to a Matrix account. When rendering, `add_threepid_success.html` is given no variable, and `add_threepid_failure.html` is given a `failure_reason`, which contains the reason for the registration failure.

HTML page templates for Single Sign-On (SSO)

Below are the templates Synapse will look for when generating pages related to SSO:

- `sso_login_idp_picker.html`: HTML page to prompt the user to choose an Identity Provider during login. This is only used if multiple SSO Identity Providers are configured. When rendering, this template is given the following variables:
 - `redirect_url`: the URL that the user will be redirected to after login.
 - `server_name`: the homeserver's name.
 - `providers`: a list of available Identity Providers. Each element is an object with the following attributes:
 - `idp_id`: unique identifier for the IdP
 - `idp_name`: user-facing name for the IdP
 - `idp_icon`: if specified in the IdP config, an MXC URI for an icon for the IdP
 - `idp_brand`: if specified in the IdP config, a textual identifier for the brand of the IdP. The rendered HTML page should contain a form which submits its results back as a GET request, with the following query parameters:
 - `redirectUrl`: the client redirect URI (ie, the `redirect_url` passed to the template)
 - `idp`: the 'idp_id' of the chosen IDP.
- `sso_auth_account_details.html`: HTML page to prompt new users to enter a userid and confirm other details. This is only shown if the SSO implementation (with any `user_mapping_provider`) does not return a localpart. When rendering, this template is given the following variables:
 - `server_name`: the homeserver's name.
 - `idp`: details of the SSO Identity Provider that the user logged in with: an object with the following attributes:
 - `idp_id`: unique identifier for the IdP
 - `idp_name`: user-facing name for the IdP
 - `idp_icon`: if specified in the IdP config, an MXC URI for an icon for the IdP
 - `idp_brand`: if specified in the IdP config, a textual identifier for the brand of the IdP
 - `user_attributes`: an object containing details about the user that we received from the IdP. May have the following attributes:
 - `display_name`: the user's display name
 - `emails`: a list of email addresses

- **localpart**: the local part of the Matrix user ID to register, if **localpart_template** is set in the mapping provider configuration (empty string if not) The template should render a form which submits the following fields:
 - **username**: the localpart of the user's chosen user id
- **sso_new_user_consent.html**: HTML page allowing the user to consent to the server's terms and conditions. This is only shown for new users, and only if **user_consent.require_at_registration** is set. When rendering, this template is given the following variables:
 - **server_name**: the homeserver's name.
 - **user_id**: the user's matrix proposed ID.
 - **user_profile.display_name**: the user's proposed display name, if any.
 - **consent_version**: the version of the terms that the user will be shown
 - **terms_url**: a link to the page showing the terms. The template should render a form which submits the following fields:
 - **accepted_version**: the version of the terms accepted by the user (ie, 'consent_version' from the input variables).
- **sso_redirect_confirm.html**: HTML page for a confirmation step before redirecting back to the client with the login token. When rendering, this template is given the following variables:
 - **redirect_url**: the URL the user is about to be redirected to.
 - **display_url**: the same as **redirect_url**, but with the query parameters stripped. The intention is to have a human-readable URL to show to users, not to use it as the final address to redirect to.
 - **server_name**: the homeserver's name.
 - **new_user**: a boolean indicating whether this is the user's first time logging in.
 - **user_id**: the user's matrix ID.
 - **user_profile.avatar_url**: an MXC URI for the user's avatar, if any. **None** if the user has not set an avatar.
 - **user_profile.display_name**: the user's display name. **None** if the user has not set a display name.
- **sso_auth_confirm.html**: HTML page which notifies the user that they are authenticating to confirm an operation on their account during the user interactive authentication process. When rendering, this template is given the following variables:
 - **redirect_url**: the URL the user is about to be redirected to.
 - **description**: the operation which the user is being asked to confirm
 - **idp**: details of the Identity Provider that we will use to confirm the user's identity: an object with the following attributes:
 - **idp_id**: unique identifier for the IdP
 - **idp_name**: user-facing name for the IdP
 - **idp_icon**: if specified in the IdP config, an MXC URI for an icon for the IdP
 - **idp_brand**: if specified in the IdP config, a textual identifier for the brand of the IdP

- **sso_auth_success.html**: HTML page shown after a successful user interactive authentication session. Note that this page must include the JavaScript which notifies of a successful authentication (see https://matrix.org/docs/spec/client_server/r0.6.0#fallback). This template has no additional variables.
- **sso_auth_bad_user.html**: HTML page shown after a user-interactive authentication session which does not map correctly onto the expected user. When rendering, this template is given the following variables:
 - `server_name`: the homeserver's name.
 - `user_id_to_verify`: the MXID of the user that we are trying to validate.
- **sso_account_deactivated.html**: HTML page shown during single sign-on if a deactivated user (according to Synapse's database) attempts to login. This template has no additional variables.
- **sso_error.html**: HTML page to display to users if something goes wrong during the OpenID Connect authentication process. When rendering, this template is given two variables:
 - `error`: the technical name of the error
 - `error_description`: a human-readable message for the error

User Authentication

Synapse supports multiple methods of authenticating users, either out-of-the-box or through custom pluggable authentication modules.

Included in Synapse is support for authenticating users via:

- A username and password.
- An email address and password.
- Single Sign-On through the SAML, Open ID Connect or CAS protocols.
- JSON Web Tokens.
- An administrator's shared secret.

Synapse can additionally be extended to support custom authentication schemes through optional "password auth provider" modules.

Single Sign-On

Synapse supports single sign-on through the SAML, Open ID Connect or CAS protocols. LDAP and other login methods are supported through first and third-party password auth provider modules.

Configuring Synapse to authenticate against an OpenID Connect provider

Synapse can be configured to use an OpenID Connect Provider (OP) for authentication, instead of its own local password database.

Any OP should work with Synapse, as long as it supports the authorization code flow. There are a few options for that:

- start a local OP. Synapse has been tested with [Hydra](#) and [Dex](#). Note that for an OP to work, it should be served under a secure (HTTPS) origin. A certificate signed with a self-signed, locally trusted CA should work. In that case, start Synapse with a `SSL_CERT_FILE` environment variable set to the path of the CA.
- set up a SaaS OP, like [Google](#), [Auth0](#) or [Okta](#). Synapse has been tested with Auth0 and Google.

It may also be possible to use other OAuth2 providers which provide the [authorization code grant type](#), such as [Github](#).

Preparing Synapse

The OpenID integration in Synapse uses the `authlib` library, which must be installed as follows:

- The relevant libraries are included in the Docker images and Debian packages provided by [matrix.org](#) so no further action is needed.
- If you installed Synapse into a virtualenv, run `/path/to/env/bin/pip install matrix-synapse[oidc]` to install the necessary dependencies.
- For other installation mechanisms, see the documentation provided by the maintainer.

To enable the OpenID integration, you should then add a section to the `oidc_providers` setting in your configuration file. See the [configuration manual](#) for some sample settings, as well as the text below for example configurations for specific providers.

OIDC Back-Channel Logout

Synapse supports receiving [OpenID Connect Back-Channel Logout](#) notifications.

This lets the OpenID Connect Provider notify Synapse when a user logs out, so that Synapse can end that user session. This feature can be enabled by setting the `backchannel_logout_enabled` property to `true` in the provider configuration, and setting the following URL as destination for Back-Channel Logout notifications in your OpenID Connect Provider: `[synapse public baseurl]/_synapse/client/oidc/backchannel_logout`

Sample configs

Here are a few configs for providers that should work with Synapse.

Microsoft Azure Active Directory

Azure AD can act as an OpenID Connect Provider. Register a new application under *App registrations* in the Azure AD management console. The RedirectURI for your application should point to your matrix server: `[synapse public baseurl]/_synapse/client/oidc/callback`

Go to *Certificates & secrets* and register a new client secret. Make note of your Directory (tenant) ID as it will be used in the Azure links. Edit your Synapse config file and change the `oidc_config` section:

```
oidc_providers:
  - idp_id: microsoft
    idp_name: Microsoft
    issuer: "https://login.microsoftonline.com/<tenant id>/v2.0"
    client_id: "<client id>"
    client_secret: "<client secret>"
    scopes: ["openid", "profile"]
    authorization_endpoint: "https://login.microsoftonline.com/<tenant id>/oauth2/v2.0/authorize"
    token_endpoint: "https://login.microsoftonline.com/<tenant id>/oauth2/v2.0/token"
    userinfo_endpoint: "https://graph.microsoft.com/oidc/userinfo"

    user_mapping_provider:
      config:
        localpart_template: "{{ user.preferred_username.split('@')[0] }}"
        display_name_template: "{{ user.name }}"
```

Apple

Configuring "Sign in with Apple" (SiWA) requires an Apple Developer account.

You will need to create a new "Services ID" for SiWA, and create and download a private key with "SiWA" enabled.

As well as the private key file, you will need:

- Client ID: the "identifier" you gave the "Services ID"
- Team ID: a 10-character ID associated with your developer account.
- Key ID: the 10-character identifier for the key.

[Apple's developer documentation](#) has more information on setting up SiWA.

The synapse config will look like this:

```
- idp_id: apple
  idp_name: Apple
  issuer: "https://appleid.apple.com"
  client_id: "your-client-id" # Set to the "identifier" for your "ServicesID"
  client_auth_method: "client_secret_post"
  client_secret_jwt_key:
    key_file: "/path/to/AuthKey_KEYIDCODE.p8" # point to your key file
    jwt_header:
      alg: ES256
      kid: "KEYIDCODE" # Set to the 10-char Key ID
    jwt_payload:
      iss: TEAMIDCODE # Set to the 10-char Team ID
    scopes: ["name", "email", "openid"]
    authorization_endpoint: https://appleid.apple.com/auth/authorize?
  response_mode=form_post
  user_mapping_provider:
    config:
      email_template: "{{ user.email }}"
```

Auth0

[Auth0](#) is a hosted SaaS IdP solution.

1. Create a regular web application for Synapse
 2. Set the Allowed Callback URLs to `[synapse public
baseurl]/_synapse/client/oidc/callback`
 3. Add a rule with any name to add the `preferred_username` claim. (See <https://auth0.com/docs/customize/rules/create-rules> for more information on how to create rules.)
- [Code sample](#)

Synapse config:

```

oidc_providers:
- idp_id: auth0
  idp_name: Auth0
  issuer: "https://your-tier.eu.auth0.com/" # TO BE FILLED
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  scopes: ["openid", "profile"]
  user_mapping_provider:
    config:
      localpart_template: "{{ user.preferred_username }}"
      display_name_template: "{{ user.name }}"

```

Authentik

[Authentik](#) is an open-source IdP solution.

1. Create a provider in Authentik, with type OAuth2/OpenID.

2. The parameters are:

- Client Type: Confidential
- JWT Algorithm: RS256
- Scopes: OpenID, Email and Profile
- RSA Key: Select any available key
- Redirect URIs: `[synapse public baseurl]/_synapse/client/oidc/callback`

3. Create an application for synapse in Authentik and link it to the provider.

4. Note the slug of your application, Client ID and Client Secret.

Note: RSA keys must be used for signing for Authentik, ECC keys do not work.

Synapse config:

```

oidc_providers:
- idp_id: authentik
  idp_name: authentik
  discover: true
  issuer: "https://your.authentik.example.org/application/o/your-app-slug/" #
  TO BE FILLED: domain and slug
  client_id: "your client id" # TO BE FILLED
  client_secret: "your client secret" # TO BE FILLED
  scopes:
    - "openid"
    - "profile"
    - "email"
  user_mapping_provider:
    config:
      localpart_template: "{{ user.preferred_username }}"
      display_name_template: "{{ user.preferred_username|capitalize }}" # TO
      BE FILLED: If your users have names in Authentik and you want those in Synapse,
      this should be replaced with user.name|capitalize.

```

Dex

[Dex](#) is a simple, open-source OpenID Connect Provider. Although it is designed to help building a full-blown provider with an external database, it can be configured with static passwords in a config file.

Follow the [Getting Started guide](#) to install Dex.

Edit `examples/config-dev.yaml` config file from the Dex repo to add a client:

```
staticClients:
- id: synapse
  secret: secret
  redirectURIs:
  - '[synapse public baseurl]/_synapse/client/oidc/callback'
  name: 'Synapse'
```

Run with `dex serve examples/config-dev.yaml`.

Synapse config:

```
oidc_providers:
- idp_id: dex
  idp_name: "My Dex server"
  skip_verification: true # This is needed as Dex is served on an insecure
  endpoint
  issuer: "http://127.0.0.1:5556/dex"
  client_id: "synapse"
  client_secret: "secret"
  scopes: ["openid", "profile"]
  user_mapping_provider:
    config:
      localpart_template: "{{ user.name }}"
      display_name_template: "{{ user.name|capitalize }}"
```

Django OAuth Toolkit

[django-oauth-toolkit](#) is a Django application providing out of the box all the endpoints, data and logic needed to add OAuth2 capabilities to your Django projects. It supports [OpenID Connect](#) too.

Configuration on Django's side:

1. Add an application: https://example.com/admin/oauth2_provider/application/add/ and choose parameters like this:

- **Redirect uris:** `https://synapse.example.com/_synapse/client/oidc/callback`
- **Client type:** `Confidential`
- **Authorization grant type:** `Authorization code`

- Algorithm: HMAC with SHA-2 256

2. You can [customize the claims](#) Django gives to synapse (optional):

► Code sample

Your synapse config is then:

```
oidc_providers:
- idp_id: django_example
  idp_name: "Django Example"
  issuer: "https://example.com/o/"
  client_id: "your-client-id" # CHANGE ME
  client_secret: "your-client-secret" # CHANGE ME
  scopes: ["openid"]
  user_profile_method: "userinfo_endpoint" # needed because oauth-toolkit
does not include user information in the authorization response
  user_mapping_provider:
    config:
      localpart_template: "{{ user.email.split('@')[0] }}"
      display_name_template: "{{ user.first_name }} {{ user.last_name }}"
      email_template: "{{ user.email }}"
```

Facebook

0. You will need a Facebook developer account. You can register for one [here](#).
1. On the [apps](#) page of the developer console, "Create App", and choose "Build Connected Experiences".
2. Once the app is created, add "Facebook Login" and choose "Web". You don't need to go through the whole form here.
3. In the left-hand menu, open "Products"/"Facebook Login"/"Settings".
 - Add `[synapse public baseurl]/_synapse/client/oidc/callback` as an OAuth Redirect URL.
4. In the left-hand menu, open "Settings/Basic". Here you can copy the "App ID" and "App Secret" for use below.

Synapse config:

```

- idp_id: facebook
  idp_name: Facebook
  idp_brand: "facebook" # optional: styling hint for clients
  discover: false
  issuer: "https://www.facebook.com"
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  scopes: ["openid", "email"]
  authorization_endpoint: "https://facebook.com/dialog/oauth"
  token_endpoint: "https://graph.facebook.com/v9.0/oauth/access_token"
  jwks_uri: "https://www.facebook.com/.well-known/oauth/openid/jwks/"
  user_mapping_provider:
    config:
      display_name_template: "{{ user.name }}"
      email_template: "{{ user.email }}"

```

Relevant documents:

- [Manually Build a Login Flow](#)
- [Using Facebook's Graph API](#)
- [Reference to the User endpoint](#)

Facebook do have an [OIDC discovery endpoint](#), but it has a [response_types_supported](#) which excludes "code" (which we rely on, and is even mentioned in their [documentation](#)), so we have to disable discovery and configure the URIs manually.

Forgejo

Forgejo is a fork of Gitea that can act as an OAuth2 provider.

The implementation of OAuth2 is improved compared to Gitea, as it provides a correctly defined [subject_claim](#) and [scopes](#).

Synapse config:

```

oidc_providers:
- idp_id: forgejo
  idp_name: Forgejo
  discover: false
  issuer: "https://your-forgejo.com/"
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  client_auth_method: client_secret_post
  scopes: ["openid", "profile", "email", "groups"]
  authorization_endpoint: "https://your-forgejo.com/login/oauth/authorize"
  token_endpoint: "https://your-forgejo.com/login/oauth/access_token"
  userinfo_endpoint: "https://your-forgejo.com/api/v1/user"
  user_mapping_provider:
    config:
      subject_claim: "sub"
      picture_claim: "picture"
      localpart_template: "{{ user.preferred_username }}"
      display_name_template: "{{ user.name }}"
      email_template: "{{ user.email }}"

```

GitHub

GitHub is a bit special as it is not an OpenID Connect compliant provider, but just a regular OAuth2 provider.

The `/user` API endpoint can be used to retrieve information on the authenticated user. As the Synapse login mechanism needs an attribute to uniquely identify users, and that endpoint does not return a `sub` property, an alternative `subject_claim` has to be set.

1. Create a new OAuth application: <https://github.com/settings/applications/new>.
2. Set the callback URL to `[synapse public baseurl]/_synapse/client/oidc/callback`.

Synapse config:

```

oidc_providers:
- idp_id: github
  idp_name: Github
  idp_brand: "github" # optional: styling hint for clients
  discover: false
  issuer: "https://github.com/"
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  authorization_endpoint: "https://github.com/login/oauth/authorize"
  token_endpoint: "https://github.com/login/oauth/access_token"
  userinfo_endpoint: "https://api.github.com/user"
  scopes: ["read:user"]
  user_mapping_provider:
    config:
      subject_claim: "id"
      localpart_template: "{{ user.login }}"
      display_name_template: "{{ user.name }}"

```

GitLab

1. Create a [new application](#).
2. Add the `read_user` and `openid` scopes.
3. Add this Callback URL: `[synapse public baseurl]/_synapse/client/oidc/callback`

Synapse config:

```
oidc_providers:
- idp_id: gitlab
  idp_name: Gitlab
  idp_brand: "gitlab" # optional: styling hint for clients
  issuer: "https://gitlab.com/"
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  client_auth_method: "client_secret_post"
  scopes: ["openid", "read_user"]
  user_profile_method: "userinfo_endpoint"
  user_mapping_provider:
    config:
      localpart_template: '{{ user.nickname }}'
      display_name_template: '{{ user.name }}'
```

Gitea

Gitea is, like Github, not an OpenID provider, but just an OAuth2 provider.

The `/user` API endpoint can be used to retrieve information on the authenticated user. As the Synapse login mechanism needs an attribute to uniquely identify users, and that endpoint does not return a `sub` property, an alternative `subject_claim` has to be set.

1. Create a new application.
2. Add this Callback URL: `[synapse public baseurl]/_synapse/client/oidc/callback`

Synapse config:

```

oidc_providers:
- idp_id: gitea
  idp_name: Gitea
  discover: false
  issuer: "https://your-gitea.com/"
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  client_auth_method: client_secret_post
  scopes: [] # Gitea doesn't support Scopes
  authorization_endpoint: "https://your-gitea.com/login/oauth/authorize"
  token_endpoint: "https://your-gitea.com/login/oauth/access_token"
  userinfo_endpoint: "https://your-gitea.com/api/v1/user"
  user_mapping_provider:
    config:
      subject_claim: "id"
      localpart_template: "{{ user.login }}"
      display_name_template: "{{ user.full_name }}"

```

Google

Google is an OpenID certified authentication and authorisation provider.

1. Set up a project in the Google API Console (see [documentation](#)).
2. Add an "OAuth Client ID" for a Web Application under "Credentials".
3. Copy the Client ID and Client Secret, and add the following to your synapse config:

```

oidc_providers:
- idp_id: google
  idp_name: Google
  idp_brand: "google" # optional: styling hint for clients
  issuer: "https://accounts.google.com/"
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  scopes: ["openid", "profile", "email"] # email is optional, read below
  user_mapping_provider:
    config:
      localpart_template: "{{ user.given_name|lower }}"
      display_name_template: "{{ user.name }}"
      email_template: "{{ user.email }}" # needs "email" in scopes above

```

4. Back in the Google console, add this Authorized redirect URI: `[synapse public baseurl]/_synapse/client/oidc/callback`.

Keycloak

Keycloak is an opensource IdP maintained by Red Hat.

Keycloak supports OIDC Back-Channel Logout, which sends logout notification to Synapse, so that Synapse users get logged out when they log out from Keycloak. This can be optionally enabled by setting `backchannel_logout_enabled` to `true` in the Synapse configuration, and by setting the "Backchannel Logout URL" in Keycloak.

Follow the [Getting Started Guide](#) to install Keycloak and set up a realm.

1. Click `Clients` in the sidebar and click `Create`

2. Fill in the fields as below:

Field	Value
Client ID	<code>synapse</code>
Client Protocol	<code>openid-connect</code>

3. Click `Save`

4. Fill in the fields as below:

Field	Value
Client ID	<code>synapse</code>
Enabled	<code>On</code>
Client Protocol	<code>openid-connect</code>
Access Type	<code>confidential</code>
Valid Redirect URIs	<code>[synapse public baseurl]/_synapse/client/oidc/callback</code>
Backchannel Logout URL (optional)	<code>[synapse public baseurl]/_synapse/client/oidc/backchannel_logout</code>
Backchannel Logout Session Required (optional)	<code>On</code>

5. Click `Save`

6. On the Credentials tab, update the fields:

Field	Value
Client Authenticator	<code>Client ID and Secret</code>

7. Click `Regenerate Secret`

8. Copy Secret

```

oidc_providers:
- idp_id: keycloak
  idp_name: "My Keycloak server"
  issuer: "https://127.0.0.1:8443/realm/{realm_name}"
  client_id: "synapse"
  client_secret: "copy secret generated from above"
  scopes: ["openid", "profile"]
  user_mapping_provider:
    config:
      localpart_template: "{{ user.preferred_username }}"
      display_name_template: "{{ user.name }}"
  backchannel_logout_enabled: true # Optional

```

LemonLDAP

LemonLDAP::NG is an open-source IdP solution.

1. Create an OpenID Connect Relying Parties in LemonLDAP::NG
2. The parameters are:

- Client ID under the basic menu of the new Relying Parties (`Options > Basic > Client ID`)
- Client secret (`Options > Basic > Client secret`)
- JWT Algorithm: RS256 within the security menu of the new Relying Parties (`Options > Security > ID Token signature algorithm` and `Options > Security > Access Token signature algorithm`)
- Scopes: OpenID, Email and Profile
- Force claims into `id_token` (`Options > Advanced > Force claims to be returned in ID Token`)
- Allowed redirection addresses for login (`Options > Basic > Allowed redirection addresses for login`): `[synapse public baseurl]/_synapse/client/oidc/callback`

Synapse config:

```

oidc_providers:
  - idp_id: lemonldap
    idp_name: lemonldap
    discover: true
    issuer: "https://auth.example.org/" # TO BE FILLED: replace with your domain
    client_id: "your client id" # TO BE FILLED
    client_secret: "your client secret" # TO BE FILLED
    scopes:
      - "openid"
      - "profile"
      - "email"
  user_mapping_provider:
    config:
      localpart_template: "{{ user.preferred_username }}"
      # TO BE FILLED: If your users have names in LemonLDAP::NG and you want
      those in Synapse, this should be replaced with user.name|capitalize or any valid
      filter.
      display_name_template: "{{ user.preferred_username|capitalize }}"

```

Mastodon

Mastodon instances provide an [OAuth API](#), allowing those instances to be used as a single sign-on provider for Synapse.

The first step is to register Synapse as an application with your Mastodon instance, using the [Create an application API](#) (see also [here](#)). There are several ways to do this, but in the example below we are using CURL.

This example assumes that:

- the Mastodon instance website URL is `https://your.mastodon.instance.url`, and
- Synapse will be registered as an app named `my_synapse_app`.

Send the following request, substituting the value of `synapse_public_baseurl` from your Synapse installation.

```

curl -d
"client_name=my_synapse_app&redirect_uris=https://[synapse_public_baseurl]/_synap
-X POST https://your.mastodon.instance.url/api/v1/apps

```

You should receive a response similar to the following. Make sure to save it.

```
{"client_id":"someclientid_123","client_secret":"someclientsecret_123","id":"1234"}
```

As the Synapse login mechanism needs an attribute to uniquely identify users, and Mastodon's endpoint does not return a `sub` property, an alternative `subject_template` has to be set. Your Synapse configuration should include the following:

```

oidc_providers:
- idp_id: my_mastodon
  idp_name: "Mastodon Instance Example"
  discover: false
  issuer: "https://your.mastodon.instance.url/@admin"
  client_id: "someclientid_123"
  client_secret: "someclientsecret_123"
  authorization_endpoint: "https://your.mastodon.instance.url/oauth/authorize"
  token_endpoint: "https://your.mastodon.instance.url/oauth/token"
  userinfo_endpoint:
    "https://your.mastodon.instance.url/api/v1/accounts/verify_credentials"
  scopes: ["read"]
  user_mapping_provider:
    config:
      subject_template: "{{ user.id }}"
      localpart_template: "{{ user.username }}"
      display_name_template: "{{ user.display_name }}"

```

Note that the fields `client_id` and `client_secret` are taken from the CURL response above.

Shibboleth with OIDC Plugin

[Shibboleth](#) is an open Standard IdP solution widely used by Universities.

1. Shibboleth needs the [OIDC Plugin](#) installed and working correctly.
2. Create a new config on the IdP Side, ensure that the `client_id` and `client_secret` are randomly generated data.

```
{
  "client_id": "SOME-CLIENT-ID",
  "client_secret": "SOME-SUPER-SECRET-SECRET",
  "response_types": ["code"],
  "grant_types": ["authorization_code"],
  "scope": "openid profile email",
  "redirect_uris": ["https://[synapse public
baseurl]/_synapse/client/oidc/callback"]
}
```

Synapse config:

```
oidc_providers:
  # Shibboleth IDP
  #
  - idp_id: shibboleth
    idp_name: "Shibboleth Login"
    discover: true
    issuer: "https://YOUR-IDP-URL.TLD"
    client_id: "YOUR_CLIENT_ID"
    client_secret: "YOUR-CLIENT-SECRET-FROM-YOUR-IDP"
    scopes: ["openid", "profile", "email"]
    allow_existing_users: true
    user_profile_method: "userinfo_endpoint"
    user_mapping_provider:
      config:
        subject_claim: "sub"
        localpart_template: "{{ user.sub.split('@')[0] }}"
        display_name_template: "{{ user.name }}"
        email_template: "{{ user.email }}"
```

Twitch

1. Setup a developer account on [Twitch](#)
2. Obtain the OAuth 2.0 credentials by [creating an app](#)
3. Add this OAuth Redirect URL: `[synapse public
baseurl]/_synapse/client/oidc/callback`

Synapse config:

```
oidc_providers:
  - idp_id: twitch
    idp_name: Twitch
    issuer: "https://id.twitch.tv/oauth2/"
    client_id: "your-client-id" # TO BE FILLED
    client_secret: "your-client-secret" # TO BE FILLED
    client_auth_method: "client_secret_post"
    user_mapping_provider:
      config:
        localpart_template: "{{ user.preferred_username }}"
        display_name_template: "{{ user.name }}"
```

Twitter

Using Twitter as an identity provider requires using Synapse 1.75.0 or later.

1. Setup a developer account on [Twitter](#)
2. Create a project & app.
3. Enable user authentication and under "Type of App" choose "Web App, Automated App or Bot".

4. Under "App info" set the callback URL to `[synapse public`

`baseurl]/_synapse/client/oidc/callback`.

5. Obtain the OAuth 2.0 credentials under the "Keys and tokens" tab, copy the "OAuth 2.0 Client ID and Client Secret"

Synapse config:

```
oidc_providers:
- idp_id: twitter
  idp_name: Twitter
  idp_brand: "twitter" # optional: styling hint for clients
  discover: false # Twitter is not OpenID compliant.
  issuer: "https://twitter.com/"
  client_id: "your-client-id" # TO BE FILLED
  client_secret: "your-client-secret" # TO BE FILLED
  pkce_method: "always"
  # offline.access providers refresh tokens, tweet.read and users.read needed
  for userinfo request.
  scopes: ["offline.access", "tweet.read", "users.read"]
  authorization_endpoint: https://twitter.com/i/oauth2/authorize
  token_endpoint: https://api.twitter.com/2/oauth2/token
  userinfo_endpoint: https://api.twitter.com/2/users/me?
  user.fields=profile_image_url
  user_mapping_provider:
    config:
      subject_template: "{{ user.data.id }}"
      localpart_template: "{{ user.data.username }}"
      display_name_template: "{{ user.data.name }}"
      picture_template: "{{ user.data.profile_image_url }}"
```

XWiki

Install [OpenID Connect Provider](#) extension in your [XWiki](#) instance.

Synapse config:

```
oidc_providers:
- idp_id: xwiki
  idp_name: "XWiki"
  issuer: "https://myxwikihost/xwiki/oidc/"
  client_id: "your-client-id" # TO BE FILLED
  client_auth_method: none
  scopes: ["openid", "profile"]
  user_profile_method: "userinfo_endpoint"
  user_mapping_provider:
    config:
      localpart_template: "{{ user.preferred_username }}"
      display_name_template: "{{ user.name }}"
```

SAML

Synapse supports authenticating users via the [Security Assertion Markup Language \(SAML\)](#) protocol natively.

Please see the `saml2_config` and `sso` sections of the [Synapse configuration file](#) for more details.

CAS

Synapse supports authenticating users via the [Central Authentication Service protocol](#) (CAS) natively.

Please see the [cas_config](#) and [sso](#) sections of the configuration manual for more details.

SSO Mapping Providers

A mapping provider is a Python class (loaded via a Python module) that works out how to map attributes of a SSO response to Matrix-specific user attributes. Details such as user ID localpart, displayname, and even avatar URLs are all things that can be mapped from talking to a SSO service.

As an example, a SSO service may return the email address "john.smith@example.com" for a user, whereas Synapse will need to figure out how to turn that into a displayname when creating a Matrix user for this individual. It may choose `John Smith`, or `Smith, John [Example.com]` or any number of variations. As each Synapse configuration may want something different, this is where SAML mapping providers come into play.

SSO mapping providers are currently supported for OpenID and SAML SSO configurations. Please see the details below for how to implement your own.

It is up to the mapping provider whether the user should be assigned a predefined Matrix ID based on the SSO attributes, or if the user should be allowed to choose their own username.

In the first case - where users are automatically allocated a Matrix ID - it is the responsibility of the mapping provider to normalise the SSO attributes and map them to a valid Matrix ID. The [specification for Matrix IDs](#) has some information about what is considered valid.

If the mapping provider does not assign a Matrix ID, then Synapse will automatically serve an HTML page allowing the user to pick their own username.

External mapping providers are provided to Synapse in the form of an external Python module. You can retrieve this module from [PyPI](#) or elsewhere, but it must be importable via Synapse (e.g. it must be in the same virtualenv as Synapse). The Synapse config is then modified to point to the mapping provider (and optionally provide additional configuration for it).

OpenID Mapping Providers

The OpenID mapping provider can be customized by editing the `oidc_providers.user_mapping_provider.module` config option.

`oidc_providers.user_mapping_provider.config` allows you to provide custom configuration options to the module. Check with the module's documentation for what options it provides (if any). The options listed by default are for the user mapping provider built in to Synapse. If using a custom module, you should comment these options out and use those specified by the module instead.

Building a Custom OpenID Mapping Provider

A custom mapping provider must specify the following methods:

- `def __init__(self, parsed_config, module_api)`
 - Arguments:
 - `parsed_config` - A configuration object that is the return value of the `parse_config` method. You should set any configuration options needed by the module here.
 - `module_api` - a `synapse.module_api.ModuleApi` object which provides the stable API available for extension modules.
- `def parse_config(config)`
 - This method should have the `@staticmethod` decoration.
 - Arguments:
 - `config` - A `dict` representing the parsed content of the `oidc_providers.user_mapping_provider.config` homeserver config option. Runs on homeserver startup. Providers should extract and validate any option values they need here.
 - Whatever is returned will be passed back to the user mapping provider module's `__init__` method during construction.
- `def get_remote_user_id(self, userinfo)`
 - Arguments:
 - `userinfo` - A `authlib.oidc.core.claims.UserInfo` object to extract user information from.
 - This method must return a string, which is the unique, immutable identifier for the user. Commonly the `sub` claim of the response.
- `async def map_user_attributes(self, userinfo, token, failures)`
 - This method must be async.
 - Arguments:
 - `userinfo` - An `authlib.oidc.core.claims.UserInfo` object to extract user information from.
 - `token` - A dictionary which includes information necessary to make further requests to the OpenID provider.
 - `failures` - An `int` that represents the amount of times the returned mxid localpart mapping has failed. This should be used to create a deduplicated mxid localpart which should be returned instead. For example, if this method returns `john.doe` as the value of `localpart` in the returned dict, and that is already taken on the homeserver, this method will be called again with the same parameters but with `failures=1`. The method should then return a different `localpart` value, such as `john.doe1`.
 - Returns a dictionary with two keys:
 - `localpart`: A string, used to generate the Matrix ID. If this is `None`, the user is prompted to pick their own username. This is only used during a user's

- first login. Once a localpart has been associated with a remote user ID (see `get_remote_user_id`) it cannot be updated.
- `confirm_localpart`: A boolean. If set to `True`, when a `localpart` string is returned from this method, Synapse will prompt the user to either accept this localpart or pick their own username. Otherwise this option has no effect. If omitted, defaults to `False`.
 - `display_name`: An optional string, the display name for the user.
 - `picture`: An optional string, the avatar url for the user.
 - `emails`: A list of strings, the email address(es) to associate with this user. If omitted, defaults to an empty list.
 - `async def get_extra_attributes(self, userinfo, token)`
 - This method must be async.
 - Arguments:
 - `userinfo` - A `authlib.oidc.core.claims.UserInfo` object to extract user information from.
 - `token` - A dictionary which includes information necessary to make further requests to the OpenID provider.
 - Returns a dictionary that is suitable to be serialized to JSON. This will be returned as part of the response during a successful login.
- Note that care should be taken to not overwrite any of the parameters usually returned as part of the [login response](#).

Default OpenID Mapping Provider

Synapse has a built-in OpenID mapping provider if a custom provider isn't specified in the config. It is located at `synapse.handlers.oidc.JinjaOidcMappingProvider`.

SAML Mapping Providers

The SAML mapping provider can be customized by editing the `saml2_config.user_mapping_provider.module` config option.

`saml2_config.user_mapping_provider.config` allows you to provide custom configuration options to the module. Check with the module's documentation for what options it provides (if any). The options listed by default are for the user mapping provider built in to Synapse. If using a custom module, you should comment these options out and use those specified by the module instead.

Building a Custom SAML Mapping Provider

A custom mapping provider must specify the following methods:

- `def __init__(self, parsed_config, module_api)`
 - Arguments:
 - `parsed_config` - A configuration object that is the return value of the `parse_config` method. You should set any configuration options needed by the module here.
 - `module_api` - a `synapse.module_api.ModuleApi` object which provides the stable API available for extension modules.
- `def parse_config(config)`
 - **This method should have the `@staticmethod` decoration.**
 - Arguments:
 - `config` - A `dict` representing the parsed content of the `saml_config.user_mapping_provider.config` homeserver config option. Runs on homeserver startup. Providers should extract and validate any option values they need here.
 - Whatever is returned will be passed back to the user mapping provider module's `__init__` method during construction.
- `def get_saml_attributes(config)`
 - **This method should have the `@staticmethod` decoration.**
 - Arguments:
 - `config` - A object resulting from a call to `parse_config`.
 - Returns a tuple of two sets. The first set equates to the SAML auth response attributes that are required for the module to function, whereas the second set consists of those attributes which can be used if available, but are not necessary.
- `def get_remote_user_id(self, saml_response, client_redirect_url)`
 - Arguments:
 - `saml_response` - A `saml2.response.AuthnResponse` object to extract user information from.
 - `client_redirect_url` - A string, the URL that the client will be redirected to.
 - This method must return a string, which is the unique, immutable identifier for the user. Commonly the `uid` claim of the response.
- `def saml_response_to_user_attributes(self, saml_response, failures, client_redirect_url)`
 - Arguments:
 - `saml_response` - A `saml2.response.AuthnResponse` object to extract user information from.
 - `failures` - An `int` that represents the amount of times the returned mxid localpart mapping has failed. This should be used to create a deduplicated mxid localpart which should be returned instead. For example, if this method returns `john.doe` as the value of `mxid_localpart` in the returned

dict, and that is already taken on the homeserver, this method will be called again with the same parameters but with failures=1. The method should then return a different `mxid_localpart` value, such as `john.doe1`.

- `client_redirect_url` - A string, the URL that the client will be redirected to.
- This method must return a dictionary, which will then be used by Synapse to build a new user. The following keys are allowed:
 - `mxid_localpart` - A string, the mxid localpart of the new user. If this is `None`, the user is prompted to pick their own username. This is only used during a user's first login. Once a localpart has been associated with a remote user ID (see `get_remote_user_id`) it cannot be updated.
 - `displayname` - The displayname of the new user. If not provided, will default to the value of `mxid_localpart`.
 - `emails` - A list of emails for the new user. If not provided, will default to an empty list.

Alternatively it can raise a `synapse.api.errors.RedirectException` to redirect the user to another page. This is useful to prompt the user for additional information, e.g. if you want them to provide their own username. It is the responsibility of the mapping provider to either redirect back to `client_redirect_url` (including any additional information) or to complete registration using methods from the `ModuleApi`.

Default SAML Mapping Provider

Synapse has a built-in SAML mapping provider if a custom provider isn't specified in the config. It is located at `synapse.handlers.saml.DefaultSamlMappingProvider`.

This page of the Synapse documentation is now deprecated. For up to date documentation on setting up or writing a password auth provider module, please see this page.

Password auth provider modules

Password auth providers offer a way for server administrators to integrate their Synapse installation with an existing authentication system.

A password auth provider is a Python class which is dynamically loaded into Synapse, and provides a number of methods by which it can integrate with the authentication system.

This document serves as a reference for those looking to implement their own password auth providers. Additionally, here is a list of known password auth provider module implementations:

- `matrix-synapse-ldap3`
- `matrix-synapse-shared-secret-auth`
- `matrix-synapse-rest-password-provider`

Required methods

Password auth provider classes must provide the following methods:

- `parse_config(config)` This method is passed the `config` object for this module from the homeserver configuration file.

It should perform any appropriate sanity checks on the provided configuration, and return an object which is then passed into `__init__`.

This method should have the `@staticmethod` decoration.

- `__init__(self, config, account_handler)`

The constructor is passed the config object returned by `parse_config`, and a `synapse.module_api.ModuleApi` object which allows the password provider to check if accounts exist and/or create new ones.

Optional methods

Password auth provider classes may optionally provide the following methods:

- `get_db_schema_files(self)`

This method, if implemented, should return an Iterable of `(name, stream)` pairs of database schema files. Each file is applied in turn at initialisation, and a record is then made in the database so that it is not re-applied on the next start.

- `get_supported_login_types(self)`

This method, if implemented, should return a `dict` mapping from a login type identifier (such as `m.login.password`) to an iterable giving the fields which must be provided by the user in the submission to the `/login` API. These fields are passed in the `login_dict` dictionary to `check_auth`.

For example, if a password auth provider wants to implement a custom login type of `com.example.custom_login`, where the client is expected to pass the fields `secret1` and `secret2`, the provider should implement this method and return the following dict:

```
{"com.example.custom_login": ("secret1", "secret2")}
```

- `check_auth(self, username, login_type, login_dict)`

This method does the real work. If implemented, it will be called for each login attempt where the login type matches one of the keys returned by `get_supported_login_types`.

It is passed the (possibly unqualified) `user` field provided by the client, the login type, and a dictionary of login secrets passed by the client.

The method should return an `Awaitable` object, which resolves to the canonical `@localpart:domain` user ID if authentication is successful, and `None` if not.

Alternatively, the `Awaitable` can resolve to a `(str, func)` tuple, in which case the second field is a callback which will be called with the result from the `/login` call (including `access_token`, `device_id`, etc.)

- `check_3pid_auth(self, medium, address, password)`

This method, if implemented, is called when a user attempts to register or log in with a third party identifier, such as email. It is passed the medium (ex. "email"), an address (ex. "jdoe@example.com") and the user's password.

The method should return an `Awaitable` object, which resolves to a `str` containing the user's (canonical) User id if authentication was successful, and `None` if not.

As with `check_auth`, the `Awaitable` may alternatively resolve to a `(user_id, callback)` tuple.

- `check_password(self, user_id, password)`

This method provides a simpler interface than `get_supported_login_types` and `check_auth` for password auth providers that just want to provide a mechanism for validating `m.login.password` logins.

If implemented, it will be called to check logins with an `m.login.password` login type. It is passed a qualified `@localpart:domain` user id, and the password provided by the user.

The method should return an `Awaitable` object, which resolves to `True` if authentication is successful, and `False` if not.

- `on_logged_out(self, user_id, device_id, access_token)`

This method, if implemented, is called when a user logs out. It is passed the qualified user ID, the ID of the deactivated device (if any: access tokens are occasionally created without an associated device ID), and the (now deactivated) access token.

It may return an `Awaitable` object; the logout request will wait for the `Awaitable` to complete, but the result is ignored.

JWT Login Type

Synapse comes with a non-standard login type to support [JSON Web Tokens](#). In general the documentation for [the login endpoint](#) is still valid (and the mechanism works similarly to the [token based login](#)).

To log in using a JSON Web Token, clients should submit a `/login` request as follows:

```
{
  "type": "org.matrix.login.jwt",
  "token": "<jwt>"
}
```

The `token` field should include the JSON web token with the following claims:

- A claim that encodes the local part of the user ID is required. By default, the `sub` (subject) claim is used, or a custom claim can be set in the configuration file.
- The expiration time (`exp`), not before time (`nbf`), and issued at (`iat`) claims are optional, but validated if present.
- The issuer (`iss`) claim is optional, but required and validated if configured.
- The audience (`aud`) claim is optional, but required and validated if configured. Providing the audience claim when not configured will cause validation to fail.

In the case that the token is not valid, the homeserver must respond with `403 Forbidden` and an error code of `M_FORBIDDEN`.

As with other login types, there are additional fields (e.g. `device_id` and `initial_device_display_name`) which can be included in the above request.

Preparing Synapse

The JSON Web Token integration in Synapse uses the [Authlib](#) library, which must be installed as follows:

- The relevant libraries are included in the Docker images and Debian packages provided by [matrix.org](#) so no further action is needed.
- If you installed Synapse into a virtualenv, run `/path/to/env/bin/pip install synapse[jwt]` to install the necessary dependencies.
- For other installation mechanisms, see the documentation provided by the maintainer.

To enable the JSON web token integration, you should then add a `jwt_config` option to your configuration file. See the [configuration manual](#) for some sample settings.

How to test JWT as a developer

Although JSON Web Tokens are typically generated from an external server, the example below uses a locally generated JWT.

1. Configure Synapse with JWT logins, note that this example uses a pre-shared secret and an algorithm of HS256:

```
jwt_config:
  enabled: true
  secret: "my-secret-token"
  algorithm: "HS256"
```

2. Generate a JSON web token:

You can use the following short Python snippet to generate a JWT protected by an HMAC. Take care that the `secret` and the algorithm given in the `header` match the entries from `jwt_config` above.

```
from authlib.jose import jwt

header = {"alg": "HS256"}
payload = {"sub": "user1", "aud": ["audience"]}
secret = "my-secret-token"
result = jwt.encode(header, payload, secret)
print(result.decode("ascii"))
```

3. Query for the login types and ensure `org.matrix.login.jwt` is there:

```
curl http://localhost:8080/_matrix/client/r0/login
```

4. Login used the generated JSON web token from above:

```
$ curl http://localhost:8082/_matrix/client/r0/login -X POST \
  --data
'{"type":"org.matrix.login.jwt","token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
--zQIc"}'
{
  "access_token": "<access token>",
  "device_id": "ACBDEFGHI",
  "home_server": "localhost:8080",
  "user_id": "@test-user:localhost:8480"
}
```

You should now be able to use the returned access token to query the client API.

Refresh Tokens

Synapse supports refresh tokens since version 1.49 (some earlier versions had support for an earlier, experimental draft of [MSC2918](#) which is not compatible).

Background and motivation

Synapse users' sessions are identified by **access tokens**; access tokens are issued to users on login. Each session gets a unique access token which identifies it; the access token must be kept secret as it grants access to the user's account.

Traditionally, these access tokens were eternally valid (at least until the user explicitly chose to log out).

In some cases, it may be desirable for these access tokens to expire so that the potential damage caused by leaking an access token is reduced. On the other hand, forcing a user to re-authenticate (log in again) often might be too much of an inconvenience.

Refresh tokens are a mechanism to avoid some of this inconvenience whilst still getting most of the benefits of short access token lifetimes. Refresh tokens are also a concept present in OAuth 2 — further reading is available [here](#).

When refresh tokens are in use, both an access token and a refresh token will be issued to users on login. The access token will expire after a predetermined amount of time, but otherwise works in the same way as before. When the access token is close to expiring (or has expired), the user's client should present the homeserver (Synapse) with the refresh token.

The homeserver will then generate a new access token and refresh token for the user and return them. The old refresh token is invalidated and can not be used again*.

Finally, refresh tokens also make it possible for sessions to be logged out if they are inactive for too long, before the session naturally ends; see the configuration guide below.

*To prevent issues if clients lose connection half-way through refreshing a token, the refresh token is only invalidated once the new access token has been used at least once. For all intents and purposes, the above simplification is sufficient.

Caveats

There are some caveats:

- If a third party gets both your access token and refresh token, they will be able to continue to enjoy access to your session.
 - This is still an improvement because you (the user) will notice when *your* session expires and you're not able to use your refresh token. That would be a giveaway that someone else has compromised your session. You would be able to log in again and terminate that session. Previously (with long-lived access tokens), a third party that has your access token could go undetected for a very long time.
- Clients need to implement support for refresh tokens in order for them to be a useful mechanism.
 - It is up to homeserver administrators if they want to issue long-lived access tokens to clients not implementing refresh tokens.
 - For compatibility, it is likely that they should, at least until client support is widespread.
 - Users with clients that support refresh tokens will still benefit from the added security; it's not possible to downgrade a session to using long-lived access tokens so this effectively gives users the choice.
 - In a closed environment where all users use known clients, this may not be an issue as the homeserver administrator can know if the clients have refresh token support. In that case, the non-refreshable access token lifetime may be set to a short duration so that a similar level of security is provided.

Configuration Guide

The following configuration options, in the `registration` section, are related:

- `session_lifetime`: maximum length of a session, even if it's refreshed. In other words, the client must log in again after this time period. In most cases, this can be unset (infinite) or set to a long time (years or months).
- `refreshable_access_token_lifetime`: lifetime of access tokens that are created by clients supporting refresh tokens. This should be short; a good value might be 5 minutes (`5m`).
- `nonrefreshable_access_token_lifetime`: lifetime of access tokens that are created by clients which don't support refresh tokens. Make this short if you want to effectively force use of refresh tokens. Make this long if you don't want to inconvenience users of clients which don't support refresh tokens (by forcing them to frequently re-authenticate using login credentials).
- `refresh_token_lifetime`: lifetime of refresh tokens. In other words, the client must refresh within this time period to maintain its session. Unless you want to log inactive sessions out, it is often fine to use a long value here or even leave it unset (infinite). Beware that making it too short will inconvenience clients that do not connect very often, including mobile clients and clients of infrequent users (by making it more

difficult for them to refresh in time, which may force them to need to re-authenticate using login credentials).

Note: All four options above only apply when tokens are created (by logging in or refreshing). Changes to these settings do not apply retroactively.

Using refresh token expiry to log out inactive sessions

If you'd like to force sessions to be logged out upon inactivity, you can enable refreshable access token expiry and refresh token expiry.

This works because a client must refresh at least once within a period of `refresh_token_lifetime` in order to maintain valid credentials to access the account.

(It's suggested that `refresh_token_lifetime` should be longer than `refreshable_access_token_lifetime` and this section assumes that to be the case for simplicity.)

Note: this will only affect sessions using refresh tokens. You may wish to set a short `nonrefreshable_access_token_lifetime` to prevent this being bypassed by clients that do not support refresh tokens.

Choosing values that guarantee permitting some inactivity

It may be desirable to permit some short periods of inactivity, for example to accommodate brief outages in client connectivity.

The following model aims to provide guidance for choosing `refresh_token_lifetime` and `refreshable_access_token_lifetime` to satisfy requirements of the form:

1. inactivity longer than `L` **MUST** cause the session to be logged out; and
2. inactivity shorter than `s` **MUST NOT** cause the session to be logged out.

This model makes the weakest assumption that all active clients will refresh as needed to maintain an active access token, but no sooner. *In reality, clients may refresh more often than this model assumes, but the above requirements will still hold.*

To satisfy the above model,

- `refresh_token_lifetime` should be set to `L`; and
- `refreshable_access_token_lifetime` should be set to `L - s`.

Overview

A captcha can be enabled on your homeserver to help prevent bots from registering accounts. Synapse currently uses Google's reCAPTCHA service which requires API keys from Google.

Getting API keys

1. Create a new site at <https://www.google.com/recaptcha/admin/create>
2. Set the label to anything you want
3. Set the type to reCAPTCHA v2 using the "I'm not a robot" Checkbox option. This is the only type of captcha that works with Synapse.
4. Add the public hostname for your server, as set in `public_baseurl` in `homeserver.yaml`, to the list of authorized domains. If you have not set `public_baseurl`, use `server_name`.
5. Agree to the terms of service and submit.
6. Copy your site key and secret key and add them to your `homeserver.yaml` configuration file

```
recaptcha_public_key: YOUR_SITE_KEY
recaptcha_private_key: YOUR_SECRET_KEY
```

7. Enable the CAPTCHA for new registrations

```
enable_registration_captcha: true
```

8. Go to the settings page for the CAPTCHA you just created
9. Uncheck the "Verify the origin of reCAPTCHA solutions" checkbox so that the captcha can be displayed in any client. If you do not disable this option then you must specify the domains of every client that is allowed to display the CAPTCHA.

Configuring IP used for auth

The reCAPTCHA API requires that the IP address of the user who solved the CAPTCHA is sent. If the client is connecting through a proxy or load balancer, it may be required to use the `X-Forwarded-For` (XFF) header instead of the origin IP address. This can be configured using the `x_forwarded` directive in the listeners section of the `homeserver.yaml` configuration file.

Registering an Application Service

The registration of new application services depends on the homeserver used. In synapse, you need to create a new configuration file for your AS and add it to the list specified under the `app_service_config_files` config option in your synapse config.

For example:

```
app_service_config_files:
- /home/matrix/.synapse/<your-AS>.yaml
```

The format of the AS configuration file is as follows:

```
id: <your-AS-id>
url: <base url of AS>
as_token: <token AS will add to requests to HS>
hs_token: <token HS will add to requests to AS>
sender_localpart: <localpart of AS user>
namespaces:
  users: # List of users we're interested in
    - exclusive: <bool>
      regex: <regex>
      group_id: <group>
    - ...
  aliases: [] # List of aliases we're interested in
  rooms: [] # List of room ids we're interested in
```

`exclusive`: If enabled, only this application service is allowed to register users in its namespace(s). `group_id`: All users of this application service are dynamically joined to this group. This is useful for e.g user organisation or flairs.

See the [spec](#) for further details on how application services work.

Server Notices

'Server Notices' are a new feature introduced in Synapse 0.30. They provide a channel whereby server administrators can send messages to users on the server.

They are used as part of communication of the server policies (see [Consent Tracking](#)), however the intention is that they may also find a use for features such as "Message of the day".

This is a feature specific to Synapse, but it uses standard Matrix communication mechanisms, so should work with any Matrix client.

User experience

When the user is first sent a server notice, they will get an invitation to a room (typically called 'Server Notices', though this is configurable in `homeserver.yaml`). They will be **unable to reject** this invitation - attempts to do so will receive an error.

Once they accept the invitation, they will see the notice message in the room history; it will appear to have come from the 'server notices user' (see below).

The user is prevented from sending any messages in this room by the power levels.

Having joined the room, the user can leave the room if they want. Subsequent server notices will then cause a new room to be created.

Synapse configuration

Server notices come from a specific user id on the server. Server administrators are free to choose the user id - something like `server` is suggested, meaning the notices will come from `@server:<your_server_name>`. Once the Server Notices user is configured, that user id becomes a special, privileged user, so administrators should ensure that **it is not already allocated**.

In order to support server notices, it is necessary to add some configuration to the `homeserver.yaml` file. In particular, you should add a `server_notices` section, which should look like this:

```
server_notices:
  system_mxid_localpart: server
  system_mxid_display_name: "Server Notices"
  system_mxid_avatar_url: "mxc://example.com/oumMVlgDnLYFaPVkExemNVVZ"
  room_name: "Server Notices"
  room_avatar_url: "mxc://example.com/oumMVlgDnLYFaPVkExemNVVZ"
  room_topic: "Room used by your server admin to notice you of important
information"
  auto_join: true
```

The only compulsory setting is `system_mxid_localpart`, which defines the user id of the Server Notices user, as above. `room_name` defines the name of the room which will be created, `room_avatar_url` its avatar and `room_topic` its topic.

`system_mxid_display_name` and `system_mxid_avatar_url` can be used to set the displayname and avatar of the Server Notices user.

`auto_join` will autojoin users to the notices room instead of sending an invite.

Sending notices

To send server notices to users you can use the [admin_api](#).

Support in Synapse for tracking agreement to server terms and conditions

Synapse 0.30 introduces support for tracking whether users have agreed to the terms and conditions set by the administrator of a server - and blocking access to the server until they have.

There are several parts to this functionality; each requires some specific configuration in `homeserver.yaml` to be enabled.

Note that various parts of the configuration and this document refer to the "privacy policy": agreement with a privacy policy is one particular use of this feature, but of course administrators can specify other terms and conditions unrelated to "privacy" per se.

Collecting policy agreement from a user

Synapse can be configured to serve the user a simple policy form with an "accept" button. Clicking "Accept" records the user's acceptance in the database and shows a success page.

To enable this, first create templates for the policy and success pages. These should be stored on the local filesystem.

These templates use the [Jinja2](#) templating language, and [docs/privacy_policy_templates](#) gives examples of the sort of thing that can be done.

Note that the templates must be stored under a name giving the language of the template - currently this must always be `en` (for "English"); internationalisation support is intended for the future.

The template for the policy itself should be versioned and named according to the version: for example `1.0.html`. The version of the policy which the user has agreed to is stored in the database.

Once the templates are in place, make the following changes to `homeserver.yaml`:

1. Add a `user_consent` section, which should look like:

```
user_consent:  
  template_dir: privacy_policy_templates  
  version: 1.0
```

`template_dir` points to the directory containing the policy templates. `version` defines the version of the policy which will be served to the user. In the example

above, Synapse will serve `privacy_policy_templates/en/1.0.html`.

2. Add a `form_secret` setting at the top level:

```
form_secret: "<unique secret>"
```

This should be set to an arbitrary secret string (try `pwgen -y 30` to generate suitable secrets).

More on what this is used for below.

3. Add `consent` wherever the `client` resource is currently enabled in the `listeners` configuration. For example:

```
listeners:
  - port: 8008
    resources:
      - names:
        - client
        - consent
```

Finally, ensure that `jinja2` is installed. If you are using a virtualenv, this should be a matter of `pip install Jinja2`. On debian, try `apt-get install python-jinja2`.

Once this is complete, and the server has been restarted, try visiting `https://<server>/_matrix/consent`. If correctly configured, this should give an error "Missing string query parameter 'u'". It is now possible to manually construct URLs where users can give their consent.

Enabling consent tracking at registration

1. Add the following to your configuration:

```
user_consent:
  require_at_registration: true
  policy_name: "Privacy Policy" # or whatever you'd like to call the policy
```

2. In your consent templates, make use of the `public_version` variable to see if an unauthenticated user is viewing the page. This is typically wrapped around the form that would be used to actually agree to the document:

```

{%
  if not public_version %}
    <!-- The variables used here are only provided when the 'u' param is
    given to the homeserver -->
    <form method="post" action="consent">
      <input type="hidden" name="v" value="{{version}}"/>
      <input type="hidden" name="u" value="{{user}}"/>
      <input type="hidden" name="h" value="{{userhmac}}"/>
      <input type="submit" value="Sure thing!"/>
    </form>
{%
  endif %}

```

3. Restart Synapse to apply the changes.

Visiting `https://<server>/_matrix/consent` should now give you a view of the privacy document. This is what users will be able to see when registering for accounts.

Constructing the consent URI

It may be useful to manually construct the "consent URI" for a given user - for instance, in order to send them an email asking them to consent. To do this, take the base

`https://<server>/_matrix/consent` URL and add the following query parameters:

- `u`: the user id of the user. This can either be a full MXID (`@user:server.com`) or just the localpart (`user`).
- `h`: hex-encoded HMAC-SHA256 of `u` using the `form_secret` as a key. It is possible to calculate this on the commandline with something like:

```
echo -n '<user>' | openssl sha256 -hmac '<form_secret>'
```

This should result in a URI which looks something like:

`https://<server>/_matrix/consent?u=<user>&h=68a152465a4d...`.

Note that not providing a `u` parameter will be interpreted as wanting to view the document from an unauthenticated perspective, such as prior to registration. Therefore, the `h` parameter is not required in this scenario. To enable this behaviour, set `require_at_registration` to `true` in your `user_consent` config.

Sending users a server notice asking them to agree to the policy

It is possible to configure Synapse to send a [server notice](#) to anybody who has not yet agreed to the current version of the policy. To do so:

- ensure that the consent resource is configured, as in the previous section
- ensure that server notices are configured, as in [the server notice documentation](#).
- Add `server_notice_content` under `user_consent` in `homeserver.yaml`. For example:

```
user_consent:
  server_notice_content:
    msgtype: m.text
    body: >-
      Please give your consent to the privacy policy at %(consent_uri)s.
```

Synapse automatically replaces the placeholder `%(consent_uri)s` with the consent uri for that user.

- ensure that `public_baseurl` is set in `homeserver.yaml`, and gives the base URI that clients use to connect to the server. (It is used to construct `consent_uri` in the server notice.)

Blocking users from using the server until they agree to the policy

Synapse can be configured to block any attempts to join rooms or send messages until the user has given their agreement to the policy. (Joining the server notices room is exempted from this).

To enable this, add `block_events_error` under `user_consent`. For example:

```
user_consent:
  block_events_error: >-
    You can't send any messages until you consent to the privacy policy at
    %(consent_uri)s.
```

Synapse automatically replaces the placeholder `%(consent_uri)s` with the consent uri for that user.

ensure that `public_baseurl` is set in `homeserver.yaml`, and gives the base URI that clients use to connect to the server. (It is used to construct `consent_uri` in the error.)

User Directory API Implementation

The user directory is maintained based on users that are 'visible' to the homeserver - i.e. ones which are local to the server and ones which any local user shares a room with.

The directory info is stored in various tables, which can sometimes get out of sync (although this is considered a bug). If this happens, for now the solution to fix it is to use the [admin API](#) and execute the job `regenerate_directory`. This should then start a background task to flush the current tables and regenerate the directory. Depending on the size of your homeserver (number of users and rooms) this can take a while.

Data model

There are five relevant tables that collectively form the "user directory". Three of them track a list of all known users. The last two (collectively called the "search tables") track which users are visible to each other.

From all of these tables we exclude three types of local user:

- support users
- appservice users
- deactivated users

A description of each table follows:

- `user_directory`. This contains the user ID, display name and avatar of each user.
 - Because there is only one directory entry per user, it is important that it only contain publicly visible information. Otherwise, this will leak the nickname or avatar used in a private room.
 - Indexed on rooms. Indexed on users.
- `user_directory_search`. To be joined to `user_directory`. It contains an extra column that enables full text search based on user IDs and display names. Different schemas for SQLite and Postgres are used.
 - Indexed on the full text search data. Indexed on users.
- `user_directory_stream_pos`. When the initial background update to populate the directory is complete, we record a stream position here. This indicates that synapse should now listen for room changes and incrementally update the directory where necessary. (See [stream positions](#).)

- `users_in_public_rooms`. Contains associations between users and the public rooms they're in. Used to determine which users are in public rooms and should be publicly visible in the directory. Both local and remote users are tracked.
- `users_who_share_private_rooms`. Rows are triples `(L, M, room_id)` where `L` is a local user and `M` is a local or remote user. `L` and `M` should be different, but this isn't enforced by a constraint.

Note that if two local users share a room then there will be two entries: `(user1, user2, !room_id)` and `(user2, user1, !room_id)`.

Configuration options

The exact way user search works can be tweaked via some server-level [configuration options](#).

The information is not repeated here, but the options are mentioned below.

Search algorithm

If `search_all_users` is `false`, then results are limited to users who:

1. Are found in the `users_in_public_rooms` table, or
2. Are found in the `users_who_share_private_rooms` where `L` is the requesting user and `M` is the search result.

Otherwise, if `search_all_users` is `true`, no such limits are placed and all users known to the server (matching the search query) will be returned.

By default, locked users are not returned. If `show_locked_users` is `true` then no filtering on the locked status of a user is done.

The user provided search term is lowercased and normalized using [NFKC](#), this treats the string as case-insensitive, canonicalizes different forms of the same text, and maps some "roughly equivalent" characters together.

The search term is then split into words:

- If [ICU](#) is available, then the system's [default locale](#) will be used to break the search term into words. (See the [installation instructions](#) for how to install ICU.)
- If unavailable, then runs of ASCII characters, numbers, underscores, and hyphens are considered words.

The queries for PostgreSQL and SQLite are detailed below, but their overall goal is to find matching users, preferring users who are "real" (e.g. not bots, not deactivated). It is assumed that real users will have a display name and avatar set.

PostgreSQL

The above words are then transformed into two queries:

1. "exact" which matches the parsed words exactly (using `to_tsquery`);
2. "prefix" which matches the parsed words as prefixes (using `to_tsquery`).

Results are composed of all rows in the `user_directory_search` table whose information matches one (or both) of these queries. Results are ordered by calculating a weighted score for each result, higher scores are returned first:

- 4x if a user ID exists.
- 1.2x if the user has a display name set.
- 1.2x if the user has an avatar set.
- 0x-3x by the full text search results using the `ts_rank_cd` function against the "exact" search query; this has four variables with the following weightings:
 - **D**: 0.1 for the user ID's domain
 - **C**: 0.1 for unused
 - **B**: 0.9 for the user's display name (or an empty string if it is not set)
 - **A**: 0.1 for the user ID's localpart
- 0x-1x by the full text search results using the `ts_rank_cd` function against the "prefix" search query. (Using the same weightings as above.)
- If `prefer_local_users` is `true`, then 2x if the user is local to the homeserver.

Note that `ts_rank_cd` returns a weight between 0 and 1. The initial weighting of all results is 1.

SQLite

Results are composed of all rows in the `user_directory_search` whose information matches the query. Results are ordered by the following information, with each subsequent column used as a tiebreaker, for each result:

1. By the `rank` of the full text search results using the `matchinfo` function. Higher ranks are returned first.
2. If `prefer_local_users` is `true`, then users local to the homeserver are returned first.
3. Users with a display name set are returned first.
4. Users with an avatar set are returned first.

Message retention policies

Synapse admins can enable support for message retention policies on their homeserver. Message retention policies exist at a room level, follow the semantics described in [MSC1763](#), and allow server and room admins to configure how long messages should be kept in a homeserver's database before being purged from it. **Please note that, as this feature isn't part of the Matrix specification yet, the use of `m.room.retention` events for per-room retention policies is to be considered as experimental. However, the use of a default message retention policy is considered a stable feature in Synapse.**

A message retention policy is mainly defined by its `max_lifetime` parameter, which defines how long a message can be kept around after it was sent to the room. If a room doesn't have a message retention policy, and there's no default one for a given server, then no message sent in that room is ever purged on that server.

MSC1763 also specifies semantics for a `min_lifetime` parameter which defines the amount of time after which an event *can* get purged (after it was sent to the room), but Synapse doesn't currently support it beyond registering it.

Both `max_lifetime` and `min_lifetime` are optional parameters.

Note that message retention policies don't apply to state events.

Once an event reaches its expiry date (defined as the time it was sent plus the value for `max_lifetime` in the room), two things happen:

- Synapse stops serving the event to clients via any endpoint.
- The message gets picked up by the next purge job (see the "Purge jobs" section) and is removed from Synapse's database.

Since purge jobs don't run continuously, this means that an event might stay in a server's database for longer than the value for `max_lifetime` in the room would allow, though hidden from clients.

Similarly, if a server (with support for message retention policies enabled) receives from another server an event that should have been purged according to its room's policy, then the receiving server will process and store that event until it's picked up by the next purge job, though it will always hide it from clients.

Synapse requires at least one message in each room, so it will never delete the last message in a room. It will, however, hide it from clients.

Server configuration

Support for this feature can be enabled and configured by adding the `retention` option in the Synapse configuration file (see [configuration manual](#)).

To enable support for message retention policies, set the setting `enabled` in this section to `true`.

Default policy

A default message retention policy is a policy defined in Synapse's configuration that is used by Synapse for every room that doesn't have a message retention policy configured in its state. This allows server admins to ensure that messages are never kept indefinitely in a server's database.

A default policy can be defined as such, by adding the `retention` option in the configuration file and adding these sub-options:

```
default_policy:  
  min_lifetime: 1d  
  max_lifetime: 1y
```

Here, `min_lifetime` and `max_lifetime` have the same meaning and level of support as previously described. They can be expressed either as a duration (using the units `s` (seconds), `m` (minutes), `h` (hours), `d` (days), `w` (weeks) and `y` (years)) or as a number of milliseconds.

Purge jobs

Purge jobs are the jobs that Synapse runs in the background to purge expired events from the database. They are only run if support for message retention policies is enabled in the server's configuration. If no configuration for purge jobs is configured by the server admin, Synapse will use a default configuration, which is described here in the [configuration manual](#).

Some server admins might want a finer control on when events are removed depending on an event's room's policy. This can be done by setting the `purge_jobs` sub-section in the `retention` section of the configuration file. An example of such configuration could be:

```
purge_jobs:
  - longest_max_lifetime: 3d
    interval: 12h
  - shortest_max_lifetime: 3d
    longest_max_lifetime: 1w
    interval: 1d
  - shortest_max_lifetime: 1w
    interval: 2d
```

In this example, we define three jobs:

- one that runs twice a day (every 12 hours) and purges events in rooms which policy's `max_lifetime` is lower or equal to 3 days.
- one that runs once a day and purges events in rooms which policy's `max_lifetime` is between 3 days and a week.
- one that runs once every 2 days and purges events in rooms which policy's `max_lifetime` is greater than a week.

Note that this example is tailored to show different configurations and features slightly more jobs than is probably necessary (in practice, a server admin would probably consider it better to replace the two last jobs with one that runs once a day and handles rooms which policy's `max_lifetime` is greater than 3 days).

Keep in mind, when configuring these jobs, that a purge job can become quite heavy on the server if it targets many rooms, therefore prefer having jobs with a low interval that target a limited set of rooms. Also make sure to include a job with no minimum and one with no maximum to make sure your configuration handles every policy.

As previously mentioned in this documentation, while a purge job that runs e.g. every day means that an expired event might stay in the database for up to a day after its expiry, Synapse hides expired events from clients as soon as they expire, so the event is not visible to local users between its expiry date and the moment it gets purged from the server's database.

Lifetime limits

Server admins can set limits on the values of `max_lifetime` to use when purging old events in a room. These limits can be defined under the `retention` option in the configuration file:

```
allowed_lifetime_min: 1d
allowed_lifetime_max: 1y
```

The limits are considered when running purge jobs. If necessary, the effective value of `max_lifetime` will be brought between `allowed_lifetime_min` and `allowed_lifetime_max` (inclusive). This means that, if the value of `max_lifetime` defined in the room's state is lower than `allowed_lifetime_min`, the value of `allowed_lifetime_min` will be used

instead. Likewise, if the value of `max_lifetime` is higher than `allowed_lifetime_max`, the value of `allowed_lifetime_max` will be used instead.

In the example above, we ensure Synapse never deletes events that are less than one day old, and that it always deletes events that are over a year old.

If a default policy is set, and its `max_lifetime` value is lower than `allowed_lifetime_min` or higher than `allowed_lifetime_max`, the same process applies.

Both parameters are optional; if one is omitted Synapse won't use it to adjust the effective value of `max_lifetime`.

Like other settings in this section, these parameters can be expressed either as a duration or as a number of milliseconds.

Room configuration

To configure a room's message retention policy, a room's admin or moderator needs to send a state event in that room with the type `m.room.retention` and the following content:

```
{  
  "max_lifetime": ...  
}
```

In this event's content, the `max_lifetime` parameter has the same meaning as previously described, and needs to be expressed in milliseconds. The event's content can also include a `min_lifetime` parameter, which has the same meaning and limited support as previously described.

Note that over every server in the room, only the ones with support for message retention policies will actually remove expired events. This support is currently not enabled by default in Synapse.

Note on reclaiming disk space

While purge jobs actually delete data from the database, the disk space used by the database might not decrease immediately on the database's host. However, even though the database engine won't free up the disk space, it will start writing new data into where the purged data was.

If you want to reclaim the freed disk space anyway and return it to the operating system, the server admin needs to run `VACUUM FULL;` (or `VACUUM;` for SQLite databases) on Synapse's database (see the related [PostgreSQL documentation](#)).

Modules

Synapse supports extending its functionality by configuring external modules.

Note: When using third-party modules, you effectively allow someone else to run custom code on your Synapse homeserver. Server admins are encouraged to verify the provenance of the modules they use on their homeserver and make sure the modules aren't running malicious code on their instance.

Using modules

To use a module on Synapse, add it to the `modules` section of the configuration file:

```
modules:
  - module: my_super_module.MySuperClass
    config:
      do_thing: true
  - module: my_other_super_module.SomeClass
    config: {}
```

Each module is defined by a path to a Python class as well as a configuration. This information for a given module should be available in the module's own documentation.

Using multiple modules

The order in which modules are listed in this section is important. When processing an action that can be handled by several modules, Synapse will always prioritise the module that appears first (i.e. is the highest in the list). This means:

- If several modules register the same callback, the callback registered by the module that appears first is used.
- If several modules try to register a handler for the same HTTP path, only the handler registered by the module that appears first is used. Handlers registered by the other module(s) are ignored and Synapse will log a warning message about them.

Note that Synapse doesn't allow multiple modules implementing authentication checkers via the password auth provider feature for the same login type with different fields. If this happens, Synapse will refuse to start.

Current status

We are currently in the process of migrating module interfaces to this system. While some interfaces might be compatible with it, others still require configuring modules in another part of Synapse's configuration file.

Currently, only the following pre-existing interfaces are compatible with this new system:

- spam checker
- third-party rules
- presence router
- password auth providers

Writing a module

A module is a Python class that uses Synapse's module API to interact with the homeserver. It can register callbacks that Synapse will call on specific operations, as well as web resources to attach to Synapse's web server.

When instantiated, a module is given its parsed configuration as well as an instance of the `synapse.module_api.ModuleApi` class. The configuration is a dictionary, and is either the output of the module's `parse_config` static method (see below), or the configuration associated with the module in Synapse's configuration file.

See the documentation for the `ModuleApi` class [here](#).

When Synapse runs with several modules configured

If Synapse is running with other modules configured, the order each module appears in within the `modules` section of the Synapse configuration file might restrict what it can or cannot register. See [this section](#) for more information.

On top of the rules listed in the link above, if a callback returns a value that should cause the current operation to fail (e.g. if a callback checking an event returns with a value that should cause the event to be denied), Synapse will fail the operation and ignore any subsequent callbacks that should have been run after this one.

The documentation for each callback mentions how Synapse behaves when multiple modules implement it.

Handling the module's configuration

A module can implement the following static method:

```
@staticmethod  
def parse_config(config: dict) -> Any
```

This method is given a dictionary resulting from parsing the YAML configuration for the module. It may modify it (for example by parsing durations expressed as strings (e.g. "5d") into milliseconds, etc.), and return the modified dictionary. It may also verify that the configuration is correct, and raise an instance of `synapse.module_api.errors.ConfigError` if not.

Registering a web resource

Modules can register web resources onto Synapse's web server using the following module API method:

```
def ModuleApi.register_web_resource(path: str, resource: IResource) -> None
```

The path is the full absolute path to register the resource at. For example, if you register a resource for the path `/_synapse/client/my_super_module/say_hello`, Synapse will serve it at `http(s)://[HS_URL]/_synapse/client/my_super_module/say_hello`. Note that Synapse does not allow registering resources for several sub-paths in the `/_matrix` namespace (such as anything under `/_matrix/client` for example). It is strongly recommended that modules register their web resources under the `/_synapse/client` namespace.

The provided resource is a Python class that implements Twisted's [IResource](#) interface (such as [Resource](#)).

Only one resource can be registered for a given path. If several modules attempt to register a resource for the same path, the module that appears first in Synapse's configuration file takes priority.

Modules **must** register their web resources in their `__init__` method.

Registering a callback

Modules can use Synapse's module API to register callbacks. Callbacks are functions that Synapse will call when performing specific actions. Callbacks must be asynchronous (unless specified otherwise), and are split in categories. A single module may implement callbacks from multiple categories, and is under no obligation to implement all callbacks from the categories it registers callbacks for.

Modules can register callbacks using one of the module API's `register_[...].callbacks` methods. The callback functions are passed to these methods as keyword arguments, with the callback name as the argument name and the function as its value. A `register_[...].callbacks` method exists for each category.

Callbacks for each category can be found on their respective page of the [Synapse documentation website](#).

Caching

Added in Synapse 1.74.0.

Modules can leverage Synapse's caching tools to manage their own cached functions. This can be helpful for modules that need to repeatedly request the same data from the database or a remote service.

Functions that need to be wrapped with a cache need to be decorated with a `@cached()` decorator (which can be imported from `synapse.module_api`) and registered with the `ModuleApi.register_cached_function` API when initialising the module. If the module needs to invalidate an entry in a cache, it needs to use the `ModuleApi.invalidate_cache` API, with the function to invalidate the cache of and the key(s) of the entry to invalidate.

Below is an example of a simple module using a cached function:

```
from typing import Any
from synapse.module_api import cached, ModuleApi

class MyModule:
    def __init__(self, config: Any, api: ModuleApi):
        self.api = api

        # Register the cached function so Synapse knows how to correctly
        # invalidate
        # entries for it.
        self.api.register_cached_function(self.get_user_from_id)

    @cached()
    async def get_department_for_user(self, user_id: str) -> str:
        """A function with a cache."""
        # Request a department from an external service.
        return await self.http_client.get_json(
            "https://int.example.com/users", {"user_id": user_id}
        )["department"]

    async def do_something_with_users(self) -> None:
        """Calls the cached function and then invalidates an entry in its
        cache."""
        user_id = "@alice@example.com"

        # Get the user. Since get_department_for_user is wrapped with a cache,
        # the return value for this user_id will be cached.
        department = await self.get_department_for_user(user_id)

        # Do something with `department`...

        # Let's say something has changed with our user, and the entry we have
        for
            # them in the cache is out of date, so we want to invalidate it.
            await self.api.invalidate_cache(self.get_department_for_user,
        (user_id,))
```

See the `cached` docstring for more details.

Spam checker callbacks

Spam checker callbacks allow module developers to implement spam mitigation actions for Synapse instances. Spam checker callbacks can be registered using the module API's `register_spam_checker_callbacks` method.

Callbacks

The available spam checker callbacks are:

`check_event_for_spam`

First introduced in Synapse v1.37.0

Changed in Synapse v1.60.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean or a string is now deprecated.

```
async def check_event_for_spam(event: "synapse.module_api.EventBase") ->
    Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes", str, bool]
```

Called when receiving an event from a client or via federation. The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.
- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) a non-`Codes` `str` to reject the operation and specify an error message. Note that clients typically will not localize the error message to the user's preferred locale.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

user_may_join_room

First introduced in Synapse v1.37.0

Changed in Synapse v1.61.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean is now deprecated.

```
async def user_may_join_room(user: str, room: str, is_invited: bool) ->
    Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes", bool]
```

Called when a user is trying to join a room. The user is represented by their Matrix user ID (e.g. `@alice:example.com`) and the room is represented by its Matrix ID (e.g. `!room:example.com`). The module is also given a boolean to indicate whether the user currently has a pending invite in the room.

This callback isn't called if the join is performed by a server administrator, or in the context of a room creation.

The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.
- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

user_may_invite

First introduced in Synapse v1.37.0

Changed in Synapse v1.62.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean is now deprecated.

```
async def user_may_invite(inviter: str, invitee: str, room_id: str) ->
    Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes", bool]
```

Called when processing an invitation, both when one is created locally or when receiving an invite over federation. Both inviter and invitee are represented by their Matrix user ID (e.g. `@alice@example.com`).

The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.
- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

`user_may_send_3pid_invite`

First introduced in Synapse v1.45.0

Changed in Synapse v1.62.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean is now deprecated.

```
async def user_may_send_3pid_invite(
    inviter: str,
    medium: str,
    address: str,
    room_id: str,
) -> Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes",
           bool]
```

Called when processing an invitation using a third-party identifier (also called a 3PID, e.g. an email address or a phone number). It is only called when a 3PID invite is created locally - not when one is received in a room over federation. If the 3PID is already associated with a Matrix ID, the spam check will go through the `user_may_invite` callback instead.

The inviter is represented by their Matrix user ID (e.g. `@alice@example.com`), and the invitee is represented by its medium (e.g. "email") and its address (e.g. `alice@example.com`). See the [Matrix specification](#) for more information regarding third-party identifiers.

For example, a call to this callback to send an invitation to the email address `alice@example.com` would look like this:

```
await user_may_send_3pid_invite(
    "@bob@example.com", # The inviter's user ID
    "email", # The medium of the 3PID to invite
    "alice@example.com", # The address of the 3PID to invite
    "!some_room@example.com", # The ID of the room to send the invite into
)
```

Note: If the third-party identifier is already associated with a matrix user ID, `user_may_invite` will be used instead.

The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.
- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

`user_may_create_room`

First introduced in Synapse v1.37.0

Changed in Synapse v1.62.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean is now deprecated.

```
async def user_may_create_room(user_id: str) ->
    Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes", bool]
```

Called when processing a room creation request.

The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.

- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

`user_may_create_room_alias`

First introduced in Synapse v1.37.0

Changed in Synapse v1.62.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean is now deprecated.

```
async def user_may_create_room_alias(user_id: str, room_alias: "synapse.module_api.RoomAlias") -> Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes", bool]
```

Called when trying to associate an alias with an existing room.

The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.
- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

user_may_publish_room

First introduced in Synapse v1.37.0

Changed in Synapse v1.62.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean is now deprecated.

```
async def user_may_publish_room(user_id: str, room_id: str) ->
    Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes", bool]
```

Called when trying to publish a room to the homeserver's public rooms directory.

The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.
- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

check_username_for_spam

First introduced in Synapse v1.37.0

```
async def check_username_for_spam(user_profile: synapse.module_api.UserProfile,
requester_id: str) -> bool
```

Called when computing search results in the user directory. The module must return a `bool` indicating whether the given user should be excluded from user directory searches. Return `True` to indicate that the user is spammy and exclude them from search results; otherwise return `False`.

The profile is represented as a dictionary with the following keys:

- `user_id: str`. The Matrix ID for this user.

- `display_name: Optional[str]`. The user's display name, or `None` if this user has not set a display name.
- `avatar_url: Optional[str]`. The `mxc://` URL to the user's avatar, or `None` if this user has not set an avatar.

The module is given a copy of the original dictionary, so modifying it from within the module cannot modify a user's profile when included in user directory search results.

The requester_id parameter is the ID of the user that called the user directory API.

If multiple modules implement this callback, they will be considered in order. If a callback returns `False`, Synapse falls through to the next one. The value of the first callback that does not return `False` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

check_registration_for_spam

First introduced in Synapse v1.37.0

```
async def check_registration_for_spam(
    email_threepid: Optional[dict],
    username: Optional[str],
    request_info: Collection[Tuple[str, str]],
    auth_provider_id: Optional[str] = None,
) -> "synapse.spam_checker_api.RegistrationBehaviour"
```

Called when registering a new user. The module must return a `RegistrationBehaviour` indicating whether the registration can go through or must be denied, or whether the user may be allowed to register but will be shadow banned.

The arguments passed to this callback are:

- `email_threepid`: The email address used for registering, if any.
- `username`: The username the user would like to register. Can be `None`, meaning that Synapse will generate one later.
- `request_info`: A collection of tuples, which first item is a user agent, and which second item is an IP address. These user agents and IP addresses are the ones that were used during the registration process.
- `auth_provider_id`: The identifier of the SSO authentication provider, if any.

If multiple modules implement this callback, they will be considered in order. If a callback returns `RegistrationBehaviour.ALLOW`, Synapse falls through to the next one. The value of the first callback that does not return `RegistrationBehaviour.ALLOW` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

check_media_file_for_spam

First introduced in Synapse v1.37.0

Changed in Synapse v1.62.0: `synapse.module_api.NOT_SPAM` and `synapse.module_api.errors.Codes` can be returned by this callback. Returning a boolean is now deprecated.

```
async def check_media_file_for_spam(
    file_wrapper: "synapse.media.media_storage.ReadableFileWrapper",
    file_info: "synapse.media._base.FileInfo",
) -> Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes", bool]
```

Called when storing a local or remote file.

The callback must return one of:

- `synapse.module_api.NOT_SPAM`, to allow the operation. Other callbacks may still decide to reject it.
- `synapse.module_api.errors.Codes` to reject the operation with an error code. In case of doubt, `synapse.module_api.errors.Codes.FORBIDDEN` is a good error code.
- (deprecated) `False`, which is the same as returning `synapse.module_api.NOT_SPAM`.
- (deprecated) `True`, which is the same as returning `synapse.module_api.errors.Codes.FORBIDDEN`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

should_drop_federated_event

First introduced in Synapse v1.60.0

```
async def should_drop_federated_event(event: "synapse.events.EventBase") -> bool
```

Called when checking whether a remote server can federate an event with us. **Returning `True` from this function will silently drop a federated event and split-brain our view of a room's DAG, and thus you shouldn't use this callback unless you know what you are doing.**

If multiple modules implement this callback, they will be considered in order. If a callback returns `False`, Synapse falls through to the next one. The value of the first callback that

does not return `False` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

check_login_for_spam

First introduced in Synapse v1.87.0

```
async def check_login_for_spam(
    user_id: str,
    device_id: Optional[str],
    initial_display_name: Optional[str],
    request_info: Collection[Tuple[Optional[str], str]],
    auth_provider_id: Optional[str] = None,
) -> Union["synapse.module_api.NOT_SPAM", "synapse.module_api.errors.Codes"]
```

Called when a user logs in.

The arguments passed to this callback are:

- `user_id`: The user ID the user is logging in with
- `device_id`: The device ID the user is re-logging into.
- `initial_display_name`: The device display name, if any.
- `request_info`: A collection of tuples, which first item is a user agent, and which second item is an IP address. These user agents and IP addresses are the ones that were used during the login process.
- `auth_provider_id`: The identifier of the SSO authentication provider, if any.

If multiple modules implement this callback, they will be considered in order. If a callback returns `synapse.module_api.NOT_SPAM`, Synapse falls through to the next one. The value of the first callback that does not return `synapse.module_api.NOT_SPAM` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

Note: This will not be called when a user registers.

Example

The example below is a module that implements the spam checker callback `check_event_for_spam` to deny any message sent by users whose Matrix user IDs are mentioned in a configured list, and registers a web resource to the path `/_synapse/client/list_spam_checker/is_evil` that returns a JSON object indicating whether the provided user appears in that list.

```
import json
from typing import Union

from twisted.web.resource import Resource
from twisted.web.server import Request

from synapse.module_api import ModuleApi

class IsUserEvilResource(Resource):
    def __init__(self, config):
        super(IsUserEvilResource, self).__init__()
        self.evil_users = config.get("evil_users") or []

    def render_GET(self, request: Request):
        user = request.args.get(b"user")[0].decode()
        request.setHeader(b"Content-Type", b"application/json")
        return json.dumps({"evil": user in self.evil_users}).encode()

class ListSpamChecker:
    def __init__(self, config: dict, api: ModuleApi):
        self.api = api
        self.evil_users = config.get("evil_users") or []

        self.api.register_spam_checker_callbacks(
            check_event_for_spam=self.check_event_for_spam,
        )

        self.api.register_web_resource(
            path="/_synapse/client/list_spam_checker/is_evil",
            resource=IsUserEvilResource(config),
        )

    @async def check_event_for_spam(self, event: "synapse.events.EventBase") ->
        Union[Literal["NOT_SPAM"], Codes]:
        if event.sender in self.evil_users:
            return Codes.FORBIDDEN
        else:
            return synapse.module_api.NOT_SPAM
```

Third party rules callbacks

Third party rules callbacks allow module developers to add extra checks to verify the validity of incoming events. Third party event rules callbacks can be registered using the module API's `register_third_party_rules_callbacks` method.

Callbacks

The available third party rules callbacks are:

`check_event_allowed`

First introduced in Synapse v1.39.0

```
async def check_event_allowed(
    event: "synapse.events.EventBase",
    state_events: "synapse.types.StateMap",
) -> Tuple[bool, Optional[dict]]
```

This callback is very experimental and can and will break without notice. Module developers are encouraged to implement `check_event_for_spam` from the spam checker category instead.

Called when processing any incoming event, with the event and a `StateMap` representing the current state of the room the event is being sent into. A `StateMap` is a dictionary that maps tuples containing an event type and a state key to the corresponding state event. For example retrieving the room's `m.room.create` event from the `state_events` argument would look like this: `state_events.get(("m.room.create", ""))`. The module must return a boolean indicating whether the event can be allowed.

Note that this callback function processes incoming events coming via federation traffic (on top of client traffic). This means denying an event might cause the local copy of the room's history to diverge from that of remote servers. This may cause federation issues in the room. It is strongly recommended to only deny events using this callback function if the sender is a local user, or in a private federation in which all servers are using the same module, with the same configuration.

If the boolean returned by the module is `True`, it may also tell Synapse to replace the event with new data by returning the new event's data as a dictionary. In order to do that, it is recommended the module calls `event.get_dict()` to get the current event as a dictionary, and modify the returned dictionary accordingly.

If `check_event_allowed` raises an exception, the module is assumed to have failed. The event will not be accepted but is not treated as explicitly rejected, either. An HTTP request causing the module check will likely result in a 500 Internal Server Error.

When the boolean returned by the module is `False`, the event is rejected. (Module developers should not use exceptions for rejection.)

Note that replacing the event only works for events sent by local users, not for events received over federation.

If multiple modules implement this callback, they will be considered in order. If a callback returns `True`, Synapse falls through to the next one. The value of the first callback that does not return `True` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

on_create_room

First introduced in Synapse v1.39.0

```
async def on_create_room(
    requester: "synapse.types.Requester",
    request_content: dict,
    is_requester_admin: bool,
) -> None
```

Called when processing a room creation request, with the `Requester` object for the user performing the request, a dictionary representing the room creation request's JSON body (see [the spec](#) for a list of possible parameters), and a boolean indicating whether the user performing the request is a server admin.

Modules can modify the `request_content` (by e.g. adding events to its `initial_state`), or deny the room's creation by raising a `module_api.errors.SynapseError`.

If multiple modules implement this callback, they will be considered in order. If a callback returns without raising an exception, Synapse falls through to the next one. The room creation will be forbidden as soon as one of the callbacks raises an exception. If this happens, Synapse will not call any of the subsequent implementations of this callback.

check_threepid_can_be_invited

First introduced in Synapse v1.39.0

```
async def check_threepid_can_be_invited(
    medium: str,
    address: str,
    state_events: "synapse.types.StateMap",
) -> bool:
```

Called when processing an invite via a third-party identifier (i.e. email or phone number). The module must return a boolean indicating whether the invite can go through.

If multiple modules implement this callback, they will be considered in order. If a callback returns `True`, Synapse falls through to the next one. The value of the first callback that does not return `True` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

check_visibility_can_be_modified

First introduced in Synapse v1.39.0

```
async def check_visibility_can_be_modified(
    room_id: str,
    state_events: "synapse.types.StateMap",
    new_visibility: str,
) -> bool:
```

Called when changing the visibility of a room in the local public room directory. The visibility is a string that's either "public" or "private". The module must return a boolean indicating whether the change can go through.

If multiple modules implement this callback, they will be considered in order. If a callback returns `True`, Synapse falls through to the next one. The value of the first callback that does not return `True` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

on_new_event

First introduced in Synapse v1.47.0

```
async def on_new_event(
    event: "synapse.events.EventBase",
    state_events: "synapse.types.StateMap",
) -> None:
```

Called after sending an event into a room. The module is passed the event, as well as the state of the room *after* the event. This means that if the event is a state event, it will be included in this state.

The state map may not be complete if Synapse hasn't yet loaded the full state of the room. This can happen for events in rooms that were just joined from a remote server.

Note that this callback is called when the event has already been processed and stored into the room, which means this callback cannot be used to deny persisting the event. To deny an incoming event, see [check_event_for_spam](#) instead.

For any given event, this callback will be called on every worker process, even if that worker will not end up acting on that event. This callback will not be called for events that are marked as rejected.

If multiple modules implement this callback, Synapse runs them all in order.

check_can_shutdown_room

First introduced in Synapse v1.55.0

```
async def check_can_shutdown_room(  
    user_id: str, room_id: str,  
) -> bool:
```

Called when an admin user requests the shutdown of a room. The module must return a boolean indicating whether the shutdown can go through. If the callback returns `False`, the shutdown will not proceed and the caller will see a `M_FORBIDDEN` error.

If multiple modules implement this callback, they will be considered in order. If a callback returns `True`, Synapse falls through to the next one. The value of the first callback that does not return `True` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

check_can_deactivate_user

First introduced in Synapse v1.55.0

```
async def check_can_deactivate_user(  
    user_id: str, by_admin: bool,  
) -> bool:
```

Called when the deactivation of a user is requested. User deactivation can be performed by an admin or the user themselves, so developers are encouraged to check the requester when implementing this callback. The module must return a boolean indicating whether the deactivation can go through. If the callback returns `False`, the deactivation will not proceed and the caller will see a `M_FORBIDDEN` error.

The module is passed two parameters, `user_id` which is the ID of the user being deactivated, and `by_admin` which is `True` if the request is made by a serve admin, and `False` otherwise.

If multiple modules implement this callback, they will be considered in order. If a callback returns `True`, Synapse falls through to the next one. The value of the first callback that does not return `True` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

on_profile_update

First introduced in Synapse v1.54.0

```
async def on_profile_update(
    user_id: str,
    new_profile: "synapse.module_api.ProfileInfo",
    by_admin: bool,
    deactivation: bool,
) -> None:
```

Called after updating a local user's profile. The update can be triggered either by the user themselves or a server admin. The update can also be triggered by a user being deactivated (in which case their display name is set to an empty string (`""`) and the avatar URL is set to `None`). The module is passed the Matrix ID of the user whose profile has been updated, their new profile, as well as a `by_admin` boolean that is `True` if the update was triggered by a server admin (and `False` otherwise), and a `deactivated` boolean that is `True` if the update is a result of the user being deactivated.

Note that the `by_admin` boolean is also `True` if the profile change happens as a result of the user logging in through Single Sign-On, or if a server admin updates their own profile.

Per-room profile changes do not trigger this callback to be called. Synapse administrators wishing this callback to be called on every profile change are encouraged to disable per-room profiles globally using the `allow_per_room_profiles` configuration setting in Synapse's configuration file. This callback is not called when registering a user, even when setting it through the `get_displayname_for_registration` module callback.

If multiple modules implement this callback, Synapse runs them all in order.

on_user_deactivation_status_changed

First introduced in Synapse v1.54.0

```
async def on_user_deactivation_status_changed(
    user_id: str, deactivated: bool, by_admin: bool
) -> None:
```

Called after deactivating a local user, or reactivating them through the admin API. The deactivation can be triggered either by the user themselves or a server admin. The module is passed the Matrix ID of the user whose status is changed, as well as a `deactivated` boolean that is `True` if the user is being deactivated and `False` if they're being reactivated, and a `by_admin` boolean that is `True` if the deactivation was triggered by a server admin (and `False` otherwise). This latter `by_admin` boolean is always `True` if the user is being reactivated, as this operation can only be performed through the admin API.

If multiple modules implement this callback, Synapse runs them all in order.

on_threepid_bind

First introduced in Synapse v1.56.0

This callback is deprecated in favour of the `on_add_user_third_party_identifier` callback, which features the same functionality. The only difference is in name.

```
async def on_threepid_bind(user_id: str, medium: str, address: str) -> None:
```

Called after creating an association between a local user and a third-party identifier (email address, phone number). The module is given the Matrix ID of the user the association is for, as well as the medium (`email` or `msisdn`) and address of the third-party identifier.

Note that this callback is *not* called after a successful association on an *identity server*.

If multiple modules implement this callback, Synapse runs them all in order.

on_add_user_third_party_identifier

First introduced in Synapse v1.79.0

```
async def on_add_user_third_party_identifier(user_id: str, medium: str, address: str) -> None:
```

Called after successfully creating an association between a user and a third-party identifier (email address, phone number). The module is given the Matrix ID of the user the association is for, as well as the medium (`email` or `msisdn`) and address of the third-party identifier (i.e. an email address).

Note that this callback is *not* called if a user attempts to bind their third-party identifier to an identity server (via a call to [POST /_matrix/client/v3/account/3pid/bind](#)).

If multiple modules implement this callback, Synapse runs them all in order.

on_remove_user_third_party_identifier

First introduced in Synapse v1.79.0

```
async def on_remove_user_third_party_identifier(user_id: str, medium: str,  
address: str) -> None:
```

Called after successfully removing an association between a user and a third-party identifier (email address, phone number). The module is given the Matrix ID of the user the association is for, as well as the medium (`email` or `msisdn`) and address of the third-party identifier (i.e. an email address).

Note that this callback is *not* called if a user attempts to unbind their third-party identifier from an identity server (via a call to `POST /_matrix/client/v3/account/3pid/unbind`).

If multiple modules implement this callback, Synapse runs them all in order.

Example

The example below is a module that implements the third-party rules callback `check_event_allowed` to censor incoming messages as dictated by a third-party service.

```
from typing import Optional, Tuple

from synapse.module_api import ModuleApi

_DEFAULT_CENSOR_ENDPOINT = "https://my-internal-service.local/censor-event"

class EventCensorer:
    def __init__(self, config: dict, api: ModuleApi):
        self.api = api
        self._endpoint = config.get("endpoint", _DEFAULT_CENSOR_ENDPOINT)

        self.api.register_third_party_rules_callbacks(
            check_event_allowed=self.check_event_allowed,
        )

    async def check_event_allowed(
        self,
        event: "synapse.events.EventBase",
        state_events: "synapse.types.StateMap",
    ) -> Tuple[bool, Optional[dict]]:
        event_dict = event.get_dict()
        new_event_content = await self.api.http_client.post_json_get_json(
            uri=self._endpoint, post_json=event_dict,
        )
        event_dict["content"] = new_event_content
        return event_dict
```

Presence router callbacks

Presence router callbacks allow module developers to define additional users which receive presence updates from local users. The additional users can be local or remote.

For example, it could be used to direct all of `@alice@example.com` (a local user)'s presence updates to `@bob:matrix.org` (a remote user), even though they don't share a room. (Note that those presence updates might not make it to `@bob:matrix.org`'s client unless a similar presence router is running on that homeserver.)

Presence router callbacks can be registered using the module API's `register_presence_router_callbacks` method.

Callbacks

The available presence router callbacks are:

`get_users_for_states`

First introduced in Synapse v1.42.0

```
async def get_users_for_states(  
    state_updates: Iterable["synapse.api.UserPresenceState"],  
) -> Dict[str, Set["synapse.api.UserPresenceState"]]
```

Requires `get_interested_users` to also be registered

Called when processing updates to the presence state of one or more users. This callback can be used to instruct the server to forward that presence state to specific users. The module must return a dictionary that maps from Matrix user IDs (which can be local or remote) to the `UserPresenceState` changes that they should be forwarded.

Synapse will then attempt to send the specified presence updates to each user when possible.

If multiple modules implement this callback, Synapse merges all the dictionaries returned by the callbacks. If multiple callbacks return a dictionary containing the same key, Synapse concatenates the sets associated with this key from each dictionary.

get_interested_users

First introduced in Synapse v1.42.0

```
async def get_interested_users(  
    user_id: str  
) -> Union[Set[str], "synapse.module_api.PRESENCE_ALL_USERS"]
```

Requires `get_users_for_states` to also be registered

Called when determining which users someone should be able to see the presence state of. This callback should return complementary results to `get_users_for_state` or the presence information may not be properly forwarded.

The callback is given the Matrix user ID for a local user that is requesting presence data and should return the Matrix user IDs of the users whose presence state they are allowed to query. The returned users can be local or remote.

Alternatively the callback can return `synapse.module_api.PRESENCE_ALL_USERS` to indicate that the user should receive updates from all known users.

If multiple modules implement this callback, they will be considered in order. Synapse calls each callback one by one, and use a concatenation of all the `set`s returned by the callbacks. If one callback returns `synapse.module_api.PRESENCE_ALL_USERS`, Synapse uses this value instead. If this happens, Synapse does not call any of the subsequent implementations of this callback.

Example

The example below is a module that implements both presence router callbacks, and ensures that `@alice@example.org` receives all presence updates from `@bob@example.com` and `@charlie:somewhere.org`, regardless of whether Alice shares a room with any of them.

```
from typing import Dict, Iterable, Set, Union

from synapse.module_api import ModuleApi


class CustomPresenceRouter:
    def __init__(self, config: dict, api: ModuleApi):
        self.api = api

        self.api.register_presence_router_callbacks(
            get_users_for_states=self.get_users_for_states,
            get_interested_users=self.get_interested_users,
        )

    async def get_users_for_states(
        self,
        state_updates: Iterable["synapse.api.UserPresenceState"],
    ) -> Dict[str, Set["synapse.api.UserPresenceState"]]:
        res = {}
        for update in state_updates:
            if (
                update.user_id == "@bob@example.com"
                or update.user_id == "@charlie:somewhere.org"
            ):
                res.setdefault("@alice@example.com", set()).add(update)

        return res

    async def get_interested_users(
        self,
        user_id: str,
    ) -> Union[Set[str], "synapse.module_api.PRESENCE_ALL_USERS"]:
        if user_id == "@alice@example.com":
            return {"@bob@example.com", "@charlie:somewhere.org"}

        return set()
```

Account validity callbacks

Account validity callbacks allow module developers to add extra steps to verify the validity on an account, i.e. see if a user can be granted access to their account on the Synapse instance. Account validity callbacks can be registered using the module API's `register_account_validity_callbacks` method.

The available account validity callbacks are:

is_user_expired

First introduced in Synapse v1.39.0

```
async def is_user_expired(user: str) -> Optional[bool]
```

Called when processing any authenticated request (except for logout requests). The module can return a `bool` to indicate whether the user has expired and should be locked out of their account, or `None` if the module wasn't able to figure it out. The user is represented by their Matrix user ID (e.g. `@alice:example.com`).

If the module returns `True`, the current request will be denied with the error code `ORG_MATRIX_EXPIRED_ACCOUNT` and the HTTP status code 403. Note that this doesn't invalidate the user's access token.

If multiple modules implement this callback, they will be considered in order. If a callback returns `None`, Synapse falls through to the next one. The value of the first callback that does not return `None` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

on_user_registration

First introduced in Synapse v1.39.0

```
async def on_user_registration(user: str) -> None
```

Called after successfully registering a user, in case the module needs to perform extra operations to keep track of them. (e.g. add them to a database table). The user is represented by their Matrix user ID.

If multiple modules implement this callback, Synapse runs them all in order.

on_user_login

First introduced in Synapse v1.98.0

```
async def on_user_login(user_id: str, auth_provider_type: str, auth_provider_id: str) -> None
```

Called after successfully login or registration of a user for cases when module needs to perform extra operations after auth. represented by their Matrix user ID.

If multiple modules implement this callback, Synapse runs them all in order.

Password auth provider callbacks

Password auth providers offer a way for server administrators to integrate their Synapse installation with an external authentication system. The callbacks can be registered by using the Module API's `register_password_auth_provider_callbacks` method.

Callbacks

auth_checkers

First introduced in Synapse v1.46.0

```
auth_checkers: Dict[Tuple[str, Tuple[str, ...]], Callable]
```

A dict mapping from tuples of a login type identifier (such as `m.login.password`) and a tuple of field names (such as `("password", "secret_thing")`) to authentication checking callbacks, which should be of the following form:

```
async def check_auth(
    user: str,
    login_type: str,
    login_dict: "synapse.module_api.JsonDict",
) -> Optional[
    Tuple[
        str,
        Optional[Callable[["synapse.module_api.LoginResponse"], Awaitable[None]]]
    ]
]
```

The login type and field names should be provided by the user in the request to the `/login` API. [The Matrix specification](#) defines some types, however user defined ones are also allowed.

The callback is passed the `user` field provided by the client (which might not be in `@username:server` form), the login type, and a dictionary of login secrets passed by the client.

If the authentication is successful, the module must return the user's Matrix ID (e.g. `@alice@example.com`) and optionally a callback to be called with the response to the `/login` request. If the module doesn't wish to return a callback, it must return `None` instead.

If the authentication is unsuccessful, the module must return `None`.

Note that the user is not automatically registered, the `register_user(...)` method of the [module API](#) can be used to lazily create users.

If multiple modules register an auth checker for the same login type but with different fields, Synapse will refuse to start.

If multiple modules register an auth checker for the same login type with the same fields, then the callbacks will be executed in order, until one returns a Matrix User ID (and optionally a callback). In that case, the return value of that callback will be accepted and subsequent callbacks will not be fired. If every callback returns `None`, then the authentication fails.

check_3pid_auth

First introduced in Synapse v1.46.0

```
async def check_3pid_auth(
    medium: str,
    address: str,
    password: str,
) -> Optional[
    Tuple[
        str,
        Optional[Callable[["synapse.module_api.LoginResponse"], Awaitable[None]]]
    ]
]
```

Called when a user attempts to register or log in with a third party identifier, such as email. It is passed the medium (eg. `email`), an address (eg. `jdoe@example.com`) and the user's password.

If the authentication is successful, the module must return the user's Matrix ID (e.g. `@alice@example.com`) and optionally a callback to be called with the response to the `/login` request. If the module doesn't wish to return a callback, it must return `None` instead.

If the authentication is unsuccessful, the module must return `None`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `None`, Synapse falls through to the next one. The value of the first callback that does not return `None` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback. If every callback returns `None`, the authentication is denied.

on_logged_out

First introduced in Synapse v1.46.0

```
async def on_logged_out(  
    user_id: str,  
    device_id: Optional[str],  
    access_token: str  
) -> None
```

Called during a logout request for a user. It is passed the qualified user ID, the ID of the deactivated device (if any: access tokens are occasionally created without an associated device ID), and the (now deactivated) access token.

Deleting the related pushers is done after calling `on_logged_out`, so you can rely on them to still be present.

If multiple modules implement this callback, Synapse runs them all in order.

get_username_for_registration

First introduced in Synapse v1.52.0

```
async def get_username_for_registration(  
    uia_results: Dict[str, Any],  
    params: Dict[str, Any],  
) -> Optional[str]
```

Called when registering a new user. The module can return a username to set for the user being registered by returning it as a string, or `None` if it doesn't wish to force a username for this user. If a username is returned, it will be used as the local part of a user's full Matrix ID (e.g. it's `alice` in `@alice:example.com`).

This callback is called once [User-Interactive Authentication](#) has been completed by the user. It is not called when registering a user via SSO. It is passed two dictionaries, which include the information that the user has provided during the registration process.

The first dictionary contains the results of the [User-Interactive Authentication](#) flow followed by the user. Its keys are the identifiers of every step involved in the flow, associated with either a boolean value indicating whether the step was correctly completed, or additional information (e.g. email address, phone number...). A list of most existing identifiers can be found in the [Matrix specification](#). Here's an example featuring all currently supported keys:

```
{
  "m.login(dummy": True, # Dummy authentication
  "m.login.terms": True, # User has accepted the terms of service for the
homeserver
  "m.login.recaptcha": True, # User has completed the recaptcha challenge
  "m.login.email.identity": { # User has provided and verified an email
address
    "medium": "email",
    "address": "alice@example.com",
    "validated_at": 1642701357084,
  },
  "m.login.msisdn": { # User has provided and verified a phone number
    "medium": "msisdn",
    "address": "33123456789",
    "validated_at": 1642701357084,
  },
  "m.login.registration_token": "sometoken", # User has registered through a
registration token
}
```

The second dictionary contains the parameters provided by the user's client in the request to `/_matrix/client/v3/register`. See the [Matrix specification](#) for a complete list of these parameters.

If the module cannot, or does not wish to, generate a username for this user, it must return `None`.

If multiple modules implement this callback, they will be considered in order. If a callback returns `None`, Synapse falls through to the next one. The value of the first callback that does not return `None` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback. If every callback returns `None`, the username provided by the user is used, if any (otherwise one is automatically generated).

get_displayname_for_registration

First introduced in Synapse v1.54.0

```
async def get_displayname_for_registration(
    uia_results: Dict[str, Any],
    params: Dict[str, Any],
) -> Optional[str]
```

Called when registering a new user. The module can return a display name to set for the user being registered by returning it as a string, or `None` if it doesn't wish to force a display name for this user.

This callback is called once [User-Interactive Authentication](#) has been completed by the user. It is not called when registering a user via SSO. It is passed two dictionaries, which include the information that the user has provided during the registration process. These

dictionaries are identical to the ones passed to `get_username_for_registration`, so refer to the documentation of this callback for more information about them.

If multiple modules implement this callback, they will be considered in order. If a callback returns `None`, Synapse falls through to the next one. The value of the first callback that does not return `None` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback. If every callback returns `None`, the username will be used (e.g. `alice` if the user being registered is `@alice:example.com`).

is_3pid_allowed

First introduced in Synapse v1.53.0

```
async def is_3pid_allowed(self, medium: str, address: str, registration: bool) -> bool
```

Called when attempting to bind a third-party identifier (i.e. an email address or a phone number). The module is given the medium of the third-party identifier (which is `email` if the identifier is an email address, or `msisdn` if the identifier is a phone number) and its address, as well as a boolean indicating whether the attempt to bind is happening as part of registering a new user. The module must return a boolean indicating whether the identifier can be allowed to be bound to an account on the local homeserver.

If multiple modules implement this callback, they will be considered in order. If a callback returns `True`, Synapse falls through to the next one. The value of the first callback that does not return `True` will be used. If this happens, Synapse will not call any of the subsequent implementations of this callback.

Example

The example module below implements authentication checkers for two different login types:

- `my.login.type`
 - Expects a `my_field` field to be sent to `/login`
 - Is checked by the method: `self.check_my_login`
- `m.login.password` (defined in [the spec](#))
 - Expects a `password` field to be sent to `/login`
 - Is checked by the method: `self.check_pass`

```
from typing import Awaitable, Callable, Optional, Tuple

import synapse
from synapse import module_api


class MyAuthProvider:
    def __init__(self, config: dict, api: module_api):

        self.api = api

        self.credentials = {
            "bob": "building",
            "@scoop:matrix.org": "digging",
        }

    api.register_password_auth_provider_callbacks(
        auth_checkers={
            ("my.login_type", ("my_field",)): self.check_my_login,
            ("m.login.password", ("password",)): self.check_pass,
        },
    )

    async def check_my_login(
        self,
        username: str,
        login_type: str,
        login_dict: "synapse.module_api.JsonDict",
    ) -> Optional[
        Tuple[
            str,
            Optional[Callable[["synapse.module_api.LoginResponse"], Awaitable[None]]],
        ]
    ]:
        if login_type != "my.login_type":
            return None

        if self.credentials.get(username) == login_dict.get("my_field"):
            return (self.api.get_qualified_user_id(username), None)

    async def check_pass(
        self,
        username: str,
        login_type: str,
        login_dict: "synapse.module_api.JsonDict",
    ) -> Optional[
        Tuple[
            str,
            Optional[Callable[["synapse.module_api.LoginResponse"], Awaitable[None]]],
        ]
    ]:
        if login_type != "m.login.password":
            return None
```

```
if self.credentials.get(username) == login_dict.get("password"):  
    return (self.api.get_qualified_user_id(username), None)
```

Background update controller callbacks

Background update controller callbacks allow module developers to control (e.g. rate-limit) how database background updates are run. A database background update is an operation Synapse runs on its database in the background after it starts. It's usually used to run database operations that would take too long if they were run at the same time as schema updates (which are run on startup) and delay Synapse's startup too much: populating a table with a big amount of data, adding an index on a big table, deleting superfluous data, etc.

Background update controller callbacks can be registered using the module API's `register_background_update_controller_callbacks` method. Only the first module (in order of appearance in Synapse's configuration file) calling this method can register background update controller callbacks, subsequent calls are ignored.

The available background update controller callbacks are:

`on_update`

First introduced in Synapse v1.49.0

```
def on_update(update_name: str, database_name: str, one_shot: bool) ->
    AsyncContextManager[int]
```

Called when about to do an iteration of a background update. The module is given the name of the update, the name of the database, and a flag to indicate whether the background update will happen in one go and may take a long time (e.g. creating indices). If this last argument is set to `False`, the update will be run in batches.

The module must return an async context manager. It will be entered before Synapse runs a background update; this should return the desired duration of the iteration, in milliseconds.

The context manager will be exited when the iteration completes. Note that the duration returned by the context manager is a target, and an iteration may take substantially longer or shorter. If the `one_shot` flag is set to `True`, the duration returned is ignored.

Note: Unlike most module callbacks in Synapse, this one is *synchronous*. This is because asynchronous operations are expected to be run by the async context manager.

This callback is required when registering any other background update controller callback.

default_batch_size

First introduced in Synapse v1.49.0

```
async def default_batch_size(update_name: str, database_name: str) -> int
```

Called before the first iteration of a background update, with the name of the update and of the database. The module must return the number of elements to process in this first iteration.

If this callback is not defined, Synapse will use a default value of 100.

min_batch_size

First introduced in Synapse v1.49.0

```
async def min_batch_size(update_name: str, database_name: str) -> int
```

Called before running a new batch for a background update, with the name of the update and of the database. The module must return an integer representing the minimum number of elements to process in this iteration. This number must be at least 1, and is used to ensure that progress is always made.

If this callback is not defined, Synapse will use a default value of 100.

Account data callbacks

Account data callbacks allow module developers to react to changes of the account data of local users. Account data callbacks can be registered using the module API's `register_account_data_callbacks` method.

Callbacks

The available account data callbacks are:

`on_account_data_updated`

First introduced in Synapse v1.57.0

```
async def on_account_data_updated(
    user_id: str,
    room_id: Optional[str],
    account_data_type: str,
    content: "synapse.module_api.JsonDict",
) -> None:
```

Called after user's account data has been updated. The module is given the Matrix ID of the user whose account data is changing, the room ID the data is associated with, the type associated with the change, as well as the new content. If the account data is not associated with a specific room, then the room ID is `None`.

This callback is triggered when new account data is added or when the data associated with a given type (and optionally room) changes. This includes deletion, since in Matrix, deleting account data consists of replacing the data associated with a given type (and optionally room) with an empty dictionary (`{}`).

Note that this doesn't trigger when changing the tags associated with a room, as these are processed separately by Synapse.

If multiple modules implement this callback, Synapse runs them all in order.

Example

The example below is a module that implements the `on_account_data_updated` callback, and sends an event to an audit room when a user changes their account data.

```
import json
import attr
from typing import Any, Dict, Optional

from synapse.module_api import JsonDict, ModuleApi
from synapse.module_api.errors import ConfigError


@attr.s(auto_attribs=True)
class CustomAccountDataConfig:
    audit_room: str
    sender: str


class CustomAccountDataModule:
    def __init__(self, config: CustomAccountDataConfig, api: ModuleApi):
        self.api = api
        self.config = config

        self.api.register_account_data_callbacks(
            on_account_data_updated=self.log_new_account_data,
        )

    @staticmethod
    def parse_config(config: Dict[str, Any]) -> CustomAccountDataConfig:
        def check_in_config(param: str):
            if param not in config:
                raise ConfigError(f"'{param}' is required")

        check_in_config("audit_room")
        check_in_config("sender")

        return CustomAccountDataConfig(
            audit_room=config["audit_room"],
            sender=config["sender"],
        )

    async def log_new_account_data(
        self,
        user_id: str,
        room_id: Optional[str],
        account_data_type: str,
        content: JsonDict,
    ) -> None:
        content_raw = json.dumps(content)
        msg_content = f"{user_id} has changed their account data for type {account_data_type} to: {content_raw}"

        if room_id is not None:
            msg_content += f" (in room {room_id})"

        await self.api.create_and_send_event_into_room(
            {
                "room_id": self.config.audit_room,
                "sender": self.config.sender,
                "type": "m.room.message",
                "content": {

```

```
        "msgtype": "m.text",
        "body": msg_content
    }
}
```

Add extra fields to client events unsigned section callbacks

First introduced in Synapse v1.96.0

This callback allows modules to add extra fields to the unsigned section of events when they get sent down to clients.

These get called *every* time an event is to be sent to clients, so care should be taken to ensure with respect to performance.

API

To register the callback, use

`register_add_extra_fields_to_unsigned_client_event_callbacks` on the `ModuleApi`.

The callback should be of the form

```
async def add_field_to_unsigned(
    event: EventBase,
) -> JsonDict:
```

where the extra fields to add to the event's unsigned section is returned. (Modules must not attempt to modify the `event` directly).

This cannot be used to alter the "core" fields in the unsigned section emitted by Synapse itself.

If multiple such callbacks try to add the same field to an event's unsigned section, the last-registered callback wins.

Porting an existing module that uses the old interface

In order to port a module that uses Synapse's old module interface, its author needs to:

- ensure the module's callbacks are all asynchronous.
- register their callbacks using one or more of the `register_[...].callbacks` methods from the `ModuleApi` class in the module's `__init__` method (see [this section](#) for more info).

Additionally, if the module is packaged with an additional web resource, the module should register this resource in its `__init__` method using the `register_web_resource` method from the `ModuleApi` class (see [this section](#) for more info).

There is no longer a `get_db_schema_files` callback provided for password auth provider modules. Any changes to the database should now be made by the module using the module API class.

The module's author should also update any example in the module's configuration to only use the new `modules` section in Synapse's configuration file (see [this section](#) for more info).

Scaling synapse via workers

For small instances it is recommended to run Synapse in the default monolith mode. For larger instances where performance is a concern it can be helpful to split out functionality into multiple separate python processes. These processes are called 'workers', and are (eventually) intended to scale horizontally independently.

Synapse's worker support is under active development and subject to change as we attempt to rapidly scale ever larger Synapse instances. However we are documenting it here to help admins needing a highly scalable Synapse instance similar to the one running matrix.org.

All processes continue to share the same database instance, and as such, workers only work with PostgreSQL-based Synapse deployments. SQLite should only be used for demo purposes and any admin considering workers should already be running PostgreSQL.

See also [Matrix.org blog post](#) for a higher level overview.

Main process/worker communication

The processes communicate with each other via a Synapse-specific protocol called 'replication' (analogous to MySQL- or Postgres-style database replication) which feeds streams of newly written data between processes so they can be kept in sync with the database state.

When configured to do so, Synapse uses a [Redis pub/sub channel](#) to send the replication stream between all configured Synapse processes. Additionally, processes may make HTTP requests to each other, primarily for operations which need to wait for a reply — such as sending an event.

All the workers and the main process connect to Redis, which relays replication commands between processes.

If Redis support is enabled Synapse will use it as a shared cache, as well as a pub/sub mechanism.

See the [Architectural diagram](#) section at the end for a visualisation of what this looks like.

Setting up workers

A Redis server is required to manage the communication between the processes. The Redis server should be installed following the normal procedure for your distribution (e.g. [apt](#)

`install redis-server` on Debian). It is safe to use an existing Redis deployment if you have one.

Once installed, check that Redis is running and accessible from the host running Synapse, for example by executing `echo PING | nc -q1 localhost 6379` and seeing a response of `+PONG`.

The appropriate dependencies must also be installed for Synapse. If using a virtualenv, these can be installed with:

```
pip install "matrix-synapse[redis]"
```

Note that these dependencies are included when synapse is installed with `pip install matrix-synapse[all]`. They are also included in the debian packages from `packages.matrix.org` and in the docker images at <https://hub.docker.com/r/ectorim/synapse/>.

To make effective use of the workers, you will need to configure an HTTP reverse-proxy such as nginx or haproxy, which will direct incoming requests to the correct worker, or to the main synapse instance. See [the reverse proxy documentation](#) for information on setting up a reverse proxy.

When using workers, each worker process has its own configuration file which contains settings specific to that worker, such as the HTTP listener that it provides (if any), logging configuration, etc.

Normally, the worker processes are configured to read from a shared configuration file as well as the worker-specific configuration files. This makes it easier to keep common configuration settings synchronised across all the processes.

The main process is somewhat special in this respect: it does not normally need its own configuration file and can take all of its configuration from the shared configuration file.

Shared configuration

Normally, only a few changes are needed to make an existing configuration file suitable for use with workers:

- First, you need to enable an "HTTP replication listener" for the main process
- Secondly, you need to enable [redis-based replication](#)
- You will need to add an `instance_map` with the `main` process defined, as well as the relevant connection information from it's HTTP `replication` listener (defined in step 1 above).
 - Note that the `host` defined is the address the worker needs to look for the `main` process at, not necessarily the same address that is bound to.

- If you are using Unix sockets for the `replication` resource, make sure to use a `path` to the socket file instead of a `port`.
- Optionally, a `shared secret` can be used to authenticate HTTP traffic between workers. For example:

```
# extend the existing `listeners` section. This defines the ports that the
# main process will listen on.
listeners:
  # The HTTP replication port
  - port: 9093
    bind_address: '127.0.0.1'
    type: http
    resources:
      - names: [replication]

  # Add a random shared secret to authenticate traffic.
  worker_replication_secret: ""

redis:
  enabled: true

instance_map:
  main:
    host: 'localhost'
    port: 9093
```

See the [configuration manual](#) for the full documentation of each option.

Under **no circumstances** should the replication listener be exposed to the public internet; replication traffic is:

- always unencrypted
- unauthenticated, unless `worker_replication_secret` is configured

Worker configuration

In the config file for each worker, you must specify:

- The type of worker (`worker_app`). The currently available worker applications are listed [below](#).
- A unique name for the worker (`worker_name`).
- If handling HTTP requests, a `worker_listeners` option with an `http` listener.
- **Synapse 1.72 and older:** if handling the `^/_matrix/client/v3/keys/upload` endpoint, the HTTP URI for the main process (`worker_main_http_uri`). This config option is no longer required and is ignored when running Synapse 1.73 and newer.

For example:

```
worker_app: synapse.app.generic_worker
worker_name: generic_worker1

worker_listeners:
- type: http
  port: 8083
  x_forwarded: true
  resources:
    - names: [client, federation]

worker_log_config: /etc/matrix-synapse/generic-worker-log.yaml
```

...is a full configuration for a generic worker instance, which will expose a plain HTTP endpoint on port 8083 separately serving various endpoints, e.g. `/sync`, which are listed below.

Obviously you should configure your reverse-proxy to route the relevant endpoints to the worker (`localhost:8083` in the above example).

Running Synapse with workers

Finally, you need to start your worker processes. This can be done with either `synctl` or your distribution's preferred service manager such as `systemd`. We recommend the use of `systemd` where available: for information on setting up `systemd` to start synapse workers, see [Systemd with Workers](#). To use `synctl`, see [Using synctl with Workers](#).

Start Synapse with Poetry

The following applies to Synapse installations that have been installed from source using `poetry`.

You can start the main Synapse process with Poetry by running the following command:

```
poetry run synapse_homeserver --config-path [your homeserver.yaml]
```

For worker setups, you can run the following command

```
poetry run synapse_worker --config-path [your homeserver.yaml] --config-path
[your worker.yaml]
```

Available worker applications

synapse.app.generic_worker

This worker can handle API requests matching the following regular expressions. These endpoints can be routed to any worker. If a worker is set up to handle a stream then, for maximum efficiency, additional endpoints should be routed to that worker: refer to the [stream writers](#) section below for further information.

```
# Sync requests
^/_matrix/client/(r0|v3)/sync$
^/_matrix/client/(api/v1|r0|v3)/events$
^/_matrix/client/(api/v1|r0|v3)/initialSync$
^/_matrix/client/(api/v1|r0|v3)/rooms/[^\/]+/initialSync$

# Federation requests
^/_matrix/federation/v1/event/
^/_matrix/federation/v1/state/
^/_matrix/federation/v1/state_ids/
^/_matrix/federation/v1/backfill/
^/_matrix/federation/v1/get_missing_events/
^/_matrix/federation/v1/publicRooms
^/_matrix/federation/v1/query/
^/_matrix/federation/v1/make_join/
^/_matrix/federation/v1/make_leave/
^/_matrix/federation/(v1|v2)/send_join/
^/_matrix/federation/(v1|v2)/send_leave/
^/_matrix/federation/v1/make_knock/
^/_matrix/federation/v1/send_knock/
^/_matrix/federation/(v1|v2)/invite/
^/_matrix/federation/v1/event_auth/
^/_matrix/federation/v1/timestamp_to_event/
^/_matrix/federation/v1/exchange_third_party_invite/
^/_matrix/federation/v1/user/devices/
^/_matrix/key/v2/query
^/_matrix/federation/v1/hierarchy/

# Inbound federation transaction request
^/_matrix/federation/v1/send/

# Client API requests
^/_matrix/client/(api/v1|r0|v3|unstable)/createRoom$
^/_matrix/client/(api/v1|r0|v3|unstable)/publicRooms$
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/joined_members$
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/context/.*$
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/members$
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/state$
^/_matrix/client/v1/rooms/.*/hierarchy$
^/_matrix/client/(v1|unstable)/rooms/.*/relations/
^/_matrix/client/v1/rooms/.*/threads$
^/_matrix/client/unstable/im.nheko.summary/summary/.*$
^/_matrix/client/(r0|v3|unstable)/account/3pid$
^/_matrix/client/(r0|v3|unstable)/account/whoami$
^/_matrix/client/(r0|v3|unstable)/devices$
^/_matrix/client/versions$
^/_matrix/client/(api/v1|r0|v3|unstable)/voip/turnServer$
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/event/
^/_matrix/client/(api/v1|r0|v3|unstable)/joined_rooms$
^/_matrix/client/v1/rooms/.*/timestamp_to_event$
^/_matrix/client/(api/v1|r0|v3|unstable/.*)/rooms/.*/aliases
^/_matrix/client/(api/v1|r0|v3|unstable)/search$
^/_matrix/client/(r0|v3|unstable)/user/.*/filter(/|$)
^/_matrix/client/(api/v1|r0|v3|unstable)/directory/room/.*$
^/_matrix/client/(r0|v3|unstable)/capabilities$
^/_matrix/client/(r0|v3|unstable)/notifications$
```

```

# Encryption requests
^/_matrix/client/(r0|v3|unstable)/keys/query$
^/_matrix/client/(r0|v3|unstable)/keys/changes$
^/_matrix/client/(r0|v3|unstable)/keys/claim$
^/_matrix/client/(r0|v3|unstable)/room_keys/
^/_matrix/client/(r0|v3|unstable)/keys/upload/

# Registration/login requests
^/_matrix/client/(api/v1|r0|v3|unstable)/login$
^/_matrix/client/(r0|v3|unstable)/register$
^/_matrix/client/(r0|v3|unstable)/register/available$
^/_matrix/client/v1/register/m.login.registration_token/validity$
^/_matrix/client/(r0|v3|unstable)/password_policy$

# Event sending requests
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/redact
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/send
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/state/
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/(join|invite|leave|ban|unban|kick)
^/_matrix/client/(api/v1|r0|v3|unstable)/join/
^/_matrix/client/(api/v1|r0|v3|unstable)/knock/
^/_matrix/client/(api/v1|r0|v3|unstable)/profile/

# User directory search requests
^/_matrix/client/(r0|v3|unstable)/user_directory/search$
```

Additionally, the following REST endpoints can be handled for GET requests:

```

^/_matrix/client/(api/v1|r0|v3|unstable)/pushrules/
^/_matrix/client/unstable/org.matrix.msc4140/delayed_events

# Account data requests
^/_matrix/client/(r0|v3|unstable)/.*/tags
^/_matrix/client/(r0|v3|unstable)/.*/account_data

# Presence requests
^/_matrix/client/(api/v1|r0|v3|unstable)/presence/
```

Pagination requests can also be handled, but all requests for a given room must be routed to the same instance. Additionally, care must be taken to ensure that the purge history admin API is not used while pagination requests for the room are in flight:

```
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/messages$
```

Additionally, the following endpoints should be included if Synapse is configured to use SSO (you only need to include the ones for whichever SSO provider you're using):

```
# for all SSO providers
^/_matrix/client/(api/v1|r0|v3|unstable)/login/sso/redirect
^/_synapse/client/pick_idp$
^/_synapse/client/pick_username
^/_synapse/client/new_user_consent$
^/_synapse/client/sso_register$

# OpenID Connect requests.
^/_synapse/client/oidc/callback$

# SAML requests.
^/_synapse/client/saml2/authn_response$

# CAS requests.
^/_matrix/client/(api/v1|r0|v3|unstable)/login/cas/ticket$
```

Ensure that all SSO logins go to a single process. For multiple workers not handling the SSO endpoints properly, see [#7530](#) and [#9427](#).

Note that a [HTTP listener](#) with `client` and `federation` `resources` must be configured in the `worker_listeners` option in the worker config.

Load balancing

It is possible to run multiple instances of this worker app, with incoming requests being load-balanced between them by the reverse-proxy. However, different endpoints have different characteristics and so admins may wish to run multiple groups of workers handling different endpoints so that load balancing can be done in different ways.

For `/sync` and `/initialSync` requests it will be more efficient if all requests from a particular user are routed to a single instance. This can be done in reverse proxy by extracting username part from the users access token.

Admins may additionally wish to separate out `/sync` requests that have a `since` query parameter from those that don't (and `/initialSync`), as requests that don't are known as "initial sync" that happens when a user logs in on a new device and can be *very* resource intensive, so isolating these requests will stop them from interfering with other users ongoing syncs.

Example `nginx` configuration snippet that handles the cases above. This is just an example and probably requires some changes according to your particular setup:

```

# Choose sync worker based on the existence of "since" query parameter
map $arg_since $sync {
    default synapse_sync;
    '' synapse_initial_sync;
}

# Extract username from access token passed as URL parameter
map $arg_access_token $accesstoken_from_urlparam {
    # Defaults to just passing back the whole accesstoken
    default $arg_access_token;
    # Try to extract username part from accesstoken URL parameter
    "~syt_(?<username>.*?)_.*" $username;
}

# Extract username from access token passed as authorization header
map $http_authorization $mxid_localpart {
    # Defaults to just passing back the whole accesstoken
    default $http_authorization;
    # Try to extract username part from accesstoken header
    "~Bearer syt_(?<username>.*?)_.*" $username;
    # if no authorization-header exist, try mapper for URL parameter
    "access_token"
        "" $accesstoken_from_urlparam;
}

upstream synapse_initial_sync {
    # Use the username mapper result for hash key
    hash $mxid_localpart consistent;
    server 127.0.0.1:8016;
    server 127.0.0.1:8036;
}

upstream synapse_sync {
    # Use the username mapper result for hash key
    hash $mxid_localpart consistent;
    server 127.0.0.1:8013;
    server 127.0.0.1:8037;
    server 127.0.0.1:8038;
    server 127.0.0.1:8039;
}

# Sync initial/normal
location ~ ^/_matrix/client/(r0|v3)/sync$ {
    proxy_pass http://$sync;
}

# Normal sync
location ~ ^/_matrix/client/(api/v1|r0|v3)/events$ {
    proxy_pass http://synapse_sync;
}

# Initial_sync
location ~ ^/_matrix/client/(api/v1|r0|v3)/initialSync$ {
    proxy_pass http://synapse_initial_sync;
}
location ~ ^/_matrix/client/(api/v1|r0|v3)/rooms/[^/]+/initialSync$ {

```

```
    proxy_pass http://synapse_initial_sync;
}
```

Federation and client requests can be balanced via simple round robin.

The inbound federation transaction request `^/_matrix/federation/v1/send/` should be balanced by source IP so that transactions from the same remote server go to the same process.

Registration/login requests can be handled separately purely to help ensure that unexpected load doesn't affect new logins and sign ups.

Finally, event sending requests can be balanced by the room ID in the URI (or the full URI, or even just round robin), the room ID is the path component after `/rooms/`. If there is a large bridge connected that is sending or may send lots of events, then a dedicated set of workers can be provisioned to limit the effects of bursts of events from that bridge on events sent by normal users.

Stream writers

Additionally, the writing of specific streams (such as events) can be moved off of the main process to a particular worker.

To enable this, the worker must have:

- An `HTTP replication` `listener` configured,
- Have a `worker_name` and be listed in the `instance_map` config.
- Have the main process declared on the `instance_map` as well.

Note: The same worker can handle multiple streams, but unless otherwise documented, each stream can only have a single writer.

For example, to move event persistence off to a dedicated worker, the shared configuration would include:

```
instance_map:
  main:
    host: localhost
    port: 8030
  event_persister1:
    host: localhost
    port: 8034

stream_writers:
  events: event_persister1
```

An example for a stream writer instance:

```

worker_app: synapse.app.generic_worker
worker_name: event_persister1

worker_listeners:
- type: http
  port: 8034
  resources:
    - names: [replication]

# Enable listener if this stream writer handles endpoints for the `typing` or
# `to_device` streams. Uses a different port to the `replication` listener to
# avoid exposing the `replication` listener publicly.
#
#- type: http
#  port: 8035
#  x_forwarded: true
#  resources:
#    - names: [client]

worker_log_config: /etc/matrix-synapse/event-persister-log.yaml

```

Some of the streams have associated endpoints which, for maximum efficiency, should be routed to the workers handling that stream. See below for the currently supported streams and the endpoints associated with them:

The `events` stream

The `events` stream experimentally supports having multiple writer workers, where load is sharded between them by room ID. Each writer is called an *event persister*. They are responsible for

- receiving new events,
- linking them to those already in the room [DAG](#),
- persisting them to the DB, and finally
- updating the events stream.

Because load is sharded in this way, you *must* restart all worker instances when adding or removing event persisters.

An `event_persister` should not be mistaken for an `event_creator`. An `event_creator` listens for requests from clients to create new events and does so. It will then pass those events over HTTP replication to any configured event persisters (or the main process if none are configured).

Note that `event_creator`s and `event_persister`s are implemented using the same [synapse.app.generic_worker](#).

An example `stream_writers` configuration with multiple writers:

```
stream_writers:
  events:
    - event_persister1
    - event_persister2
```

The **typing** stream

The following endpoints should be routed directly to the worker configured as the stream writer for the **typing** stream:

```
^/_matrix/client/(api/v1|r0|v3|unstable)/rooms/.*/typing
```

The **to_device** stream

The following endpoints should be routed directly to the worker configured as the stream writer for the **to_device** stream:

```
^/_matrix/client/(r0|v3|unstable)/sendToDevice/
```

The **account_data** stream

The following endpoints should be routed directly to the worker configured as the stream writer for the **account_data** stream:

```
^/_matrix/client/(r0|v3|unstable)/.*/tags
^/_matrix/client/(r0|v3|unstable)/.*/account_data
```

The **receipts** stream

The following endpoints should be routed directly to the worker configured as the stream writer for the **receipts** stream:

```
^/_matrix/client/(r0|v3|unstable)/rooms/.*/receipt
^/_matrix/client/(r0|v3|unstable)/rooms/.*/read_markers
```

The **presence** stream

The following endpoints should be routed directly to the worker configured as the stream writer for the **presence** stream:

```
^/_matrix/client/(api/v1|r0|v3|unstable)/presence/
```

The **push_rules** stream

The following endpoints should be routed directly to the worker configured as the stream writer for the **push_rules** stream:

```
^/_matrix/client/(api/v1|r0|v3|unstable)/pushrules/
```

Restrict outbound federation traffic to a specific set of workers

The `outbound_federation_restricted_to` configuration is useful to make sure outbound federation traffic only goes through a specified subset of workers. This allows you to set more strict access controls (like a firewall) for all workers and only allow the `federation_sender`'s to contact the outside world.

```
instance_map:
  main:
    host: localhost
    port: 8030
  federation_sender1:
    host: localhost
    port: 8034

outbound_federation_restricted_to:
  - federation_sender1

worker_replication_secret: "secret_secret"
```

Background tasks

There is also support for moving background tasks to a separate worker. Background tasks are run periodically or started via replication. Exactly which tasks are configured to run depends on your Synapse configuration (e.g. if stats is enabled). This worker doesn't handle any REST endpoints itself.

To enable this, the worker must have a unique `worker_name` and can be configured to run background tasks. For example, to move background tasks to a dedicated worker, the shared configuration would include:

```
run_background_tasks_on: background_worker
```

You might also wish to investigate the `update_user_directory_from_worker` and `media_instance_running_background_jobs` settings.

An example for a dedicated background worker instance:

```
worker_app: synapse.app.generic_worker
worker_name: background_worker

worker_log_config: /etc/matrix-synapse/background-worker-log.yaml
```

Updating the User Directory

You can designate one generic worker to update the user directory.

Specify its name in the [shared configuration](#) as follows:

```
update_user_directory_from_worker: worker_name
```

This work cannot be load-balanced; please ensure the main process is restarted after setting this option in the shared configuration!

User directory updates allow REST endpoints matching the following regular expressions to work:

```
^/_matrix/client/(r0|v3|unstable)/user_directory/search$
```

The above endpoints can be routed to any worker, though you may choose to route it to the chosen user directory worker.

This style of configuration supersedes the legacy `synapse.app.user_dir` worker application type.

Notifying Application Services

You can designate one generic worker to send output traffic to Application Services. Doesn't handle any REST endpoints itself, but you should specify its name in the [shared configuration](#) as follows:

```
notify_appservices_from_worker: worker_name
```

This work cannot be load-balanced; please ensure the main process is restarted after setting this option in the shared configuration!

This style of configuration supersedes the legacy `synapse.app.appservice` worker application type.

Push Notifications

You can designate generic workers to send push notifications to a [push gateway](#) such as [sygnal](#) and email.

This will stop the main process sending push notifications.

The workers responsible for sending push notifications can be defined using the `pusher_instances` option. For example:

```
pusher_instances:
  - pusher_worker1
  - pusher_worker2
```

Multiple workers can be added to this map, in which case the work is balanced across them. Ensure the main process and all pusher workers are restarted after changing this option.

These workers don't need to accept incoming HTTP requests to send push notifications, so no additional reverse proxy configuration is required for pusher workers.

This style of configuration supersedes the legacy `synapse.app.pusher` worker application type.

`synapse.app.pusher`

It is likely this option will be deprecated in the future and is not recommended for new installations. Instead, use `synapse.app.generic_worker` with the `pusher_instances`.

Handles sending push notifications to syignal and email. Doesn't handle any REST endpoints itself, but you should set `start_pushers: false` in the shared configuration file to stop the main synapse sending push notifications.

To run multiple instances at once the `pusher_instances` option should list all pusher instances by their `worker_name`, e.g.:

```
start_pushers: false
pusher_instances:
  - pusher_worker1
  - pusher_worker2
```

An example for a pusher instance:

```
worker_app: synapse.app.pusher
worker_name: pusher_worker1

worker_log_config: /etc/matrix-synapse/pusher-worker-log.yaml
```

`synapse.app.appservice`

Deprecated as of Synapse v1.59. Use `synapse.app.generic_worker` with the `notify_appservices_from_worker` option instead.

Handles sending output traffic to Application Services. Doesn't handle any REST endpoints itself, but you should set `notify_appservices: False` in the shared configuration file to stop the main synapse sending appservice notifications.

Note this worker cannot be load-balanced: only one instance should be active.

synapse.app.federation_sender

It is likely this option will be deprecated in the future and not recommended for new installations. Instead, use `synapse.app.generic_worker` with the `federation_sender_instances`.

Handles sending federation traffic to other servers. Doesn't handle any REST endpoints itself, but you should set `send_federation: false` in the shared configuration file to stop the main synapse sending this traffic.

If running multiple federation senders then you must list each instance in the `federation_sender_instances` option by their `worker_name`. All instances must be stopped and started when adding or removing instances. For example:

```
send_federation: false
federation_sender_instances:
  - federation_sender1
  - federation_sender2
```

An example for a federation sender instance:

```
worker_app: synapse.app.federation_sender
worker_name: federation_sender1

worker_log_config: /etc/matrix-synapse/federation-sender-log.yaml
```

synapse.app.media_repository

Handles the media repository. It can handle all endpoints starting with:

```
/_matrix/media/
/_matrix/client/v1/media/
/_matrix/federation/v1/media/
```

... and the following regular expressions matching media-specific administration APIs:

```
^/_synapse/admin/v1/purge_media_cache$
^/_synapse/admin/v1/room/.*/media.*$
^/_synapse/admin/v1/user/.*/media.*$
^/_synapse/admin/v1/media/.*$
^/_synapse/admin/v1/quarantine_media/.*$
^/_synapse/admin/v1/users/.*/media$
```

You should also set `enable_media_repo: False` in the shared configuration file to stop the main synapse running background jobs related to managing the media repository. Note that

doing so will prevent the main process from being able to handle the above endpoints.

In the `media_repository` worker configuration file, configure the [HTTP listener](#) to expose the `media` resource. For example:

```
worker_app: synapse.app.media_repository
worker_name: media_worker

worker_listeners:
- type: http
  port: 8085
  x_forwarded: true
  resources:
  - names: [media]

worker_log_config: /etc/matrix-synapse/media-worker-log.yaml
```

Note that if running multiple media repositories they must be on the same server and you must specify a single instance to run the background tasks in the [shared configuration](#), e.g.:

```
media_instance_running_background_jobs: "media-repository-1"
```

Note that if a reverse proxy is used, then `/_matrix/media/` must be routed for both inbound client and federation requests (if they are handled separately).

`synapse.app.user_dir`

Deprecated as of Synapse v1.59. Use `synapse.app.generic_worker` with the `update_user_directory_from_worker` option instead.

Handles searches in the user directory. It can handle REST endpoints matching the following regular expressions:

```
^/_matrix/client/(r0|v3|unstable)/user_directory/search$
```

When using this worker you must also set `update_user_directory: false` in the shared configuration file to stop the main synapse running background jobs related to updating the user directory.

Above endpoint is not *required* to be routed to this worker. By default, `update_user_directory` is set to `true`, which means the main process will handle updates. All workers configured with `client` can handle the above endpoint as long as either this worker or the main process are configured to handle it, and are online.

If `update_user_directory` is set to `false`, and this worker is not running, the above endpoint may give outdated results.

Historical apps

The following used to be separate worker application types, but are now equivalent to `synapse.app.generic_worker`:

- `synapse.app.client_reader`
- `synapse.app.event_creator`
- `synapse.app.federation_reader`
- `synapse.app.federation_sender`
- `synapse.app.frontend_proxy`
- `synapse.app.pusher`
- `synapse.app.synchrotron`

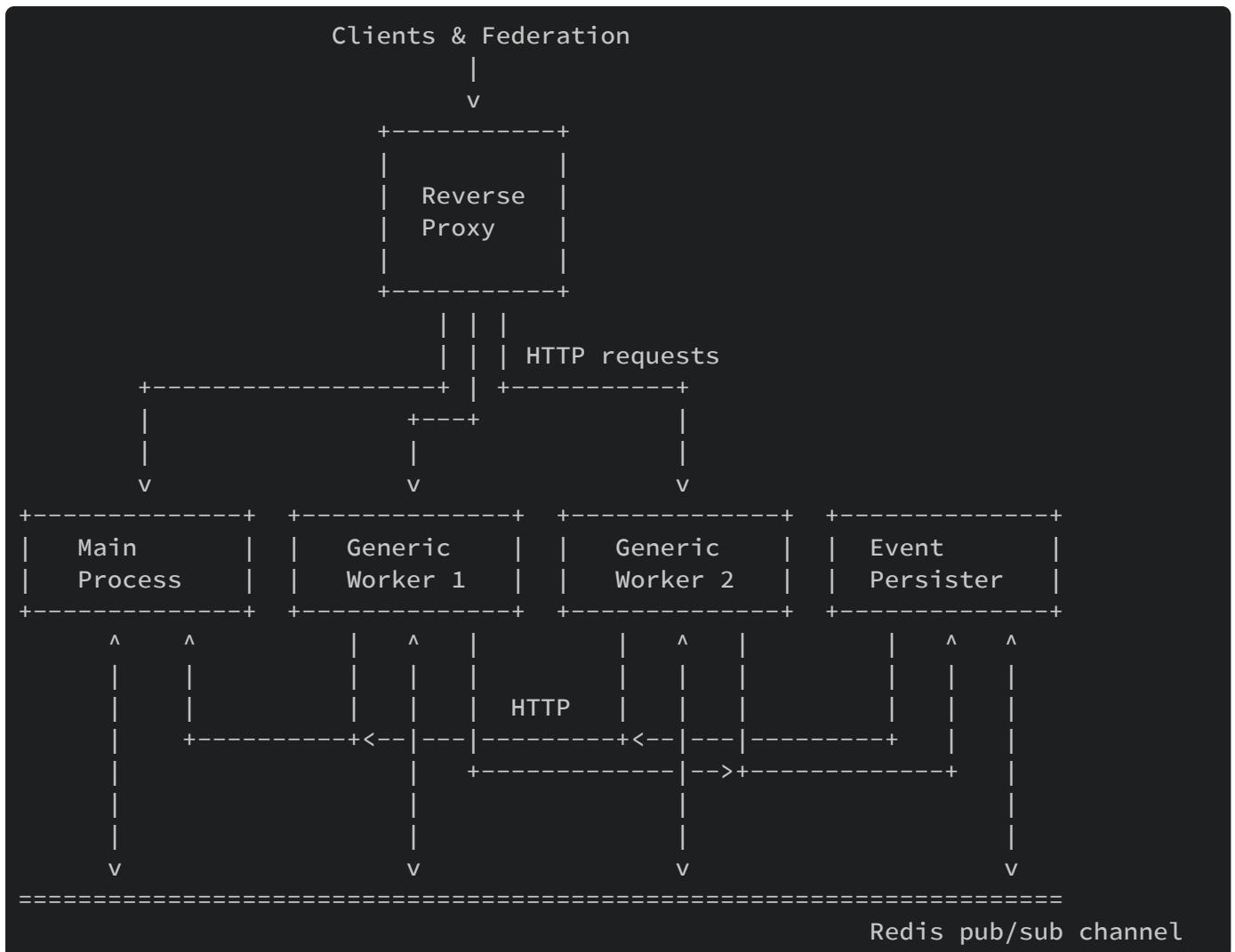
Migration from old config

A main change that has occurred is the merging of worker apps into `synapse.app.generic_worker`. This change is backwards compatible and so no changes to the config are required.

To migrate apps to use `synapse.app.generic_worker` simply update the `worker_app` option in the worker configs, and where workers are started (e.g. in systemd service files, but not required for synctl).

Architectural diagram

The following shows an example setup using Redis and a reverse proxy:



Using synctl with workers

If you want to use `synctl` to manage your synapse processes, you will need to create an additional configuration file for the main synapse process. That configuration should look like this:

```
worker_app: synapse.app.homeserver
```

Additionally, each worker app must be configured with the name of a "pid file", to which it will write its process ID when it starts. For example, for a synchrotron, you might write:

```
worker_pid_file: /home/matrix/synapse/worker1.pid
```

Finally, to actually run your worker-based synapse, you must pass `synctl` the `-a` commandline option to tell it to operate on all the worker configurations found in the given directory, e.g.:

```
synctl -a $CONFIG/workers start
```

Currently one should always restart all workers when restarting or upgrading synapse, unless you explicitly know it's safe not to. For instance, restarting synapse without restarting all the synchrotrons may result in broken typing notifications.

To manipulate a specific worker, you pass the `-w` option to `synctl`:

```
synctl -w $CONFIG/workers/worker1.yaml restart
```

Setting up Synapse with Workers and Systemd

This is a setup for managing synapse with systemd, including support for managing workers. It provides a `matrix-synapse` service for the master, as well as a `matrix-synapse-worker@` service template for any workers you require. Additionally, to group the required services, it sets up a `matrix-synapse.target`.

See the folder `system` for the systemd unit files.

The folder `workers` contains an example configuration for the `generic_worker` worker.

Synapse configuration files

See [the worker documentation](#) for information on how to set up the configuration files and reverse-proxy correctly. Below is a sample `generic_worker` worker configuration file.

```
worker_app: synapse.app.generic_worker
worker_name: generic_worker1

worker_listeners:
  - type: http
    port: 8083
    x_forwarded: true
    resources:
      - names: [client, federation]

worker_log_config: /etc/matrix-synapse/generic-worker-log.yaml
```

Systemd manages daemonization itself, so ensure that none of the configuration files set either `daemonize` or `worker_daemonize`.

The config files of all workers are expected to be located in `/etc/matrix-synapse/workers`. If you want to use a different location, edit the provided `*.service` files accordingly.

There is no need for a separate configuration file for the master process.

Set up

1. Adjust synapse configuration files as above.
2. Copy the `*.service` and `*.target` files in `system` to `/etc/systemd/system`.
3. Run `systemctl daemon-reload` to tell systemd to load the new unit files.

4. Run `systemctl enable matrix-synapse.service`. This will configure the synapse master process to be started as part of the `matrix-synapse.target` target.
5. For each worker process to be enabled, run `systemctl enable matrix-synapse-worker@<worker_name>.service`. For each `<worker_name>`, there should be a corresponding configuration file. `/etc/matrix-synapse/workers/<worker_name>.yaml`.
6. Start all the synapse processes with `systemctl start matrix-synapse.target`.
7. Tell systemd to start synapse on boot with `systemctl enable matrix-synapse.target`.

Usage

Once the services are correctly set up, you can use the following commands to manage your synapse installation:

```
# Restart Synapse master and all workers
systemctl restart matrix-synapse.target

# Stop Synapse and all workers
systemctl stop matrix-synapse.target

# Restart the master alone
systemctl start matrix-synapse.service

# Restart a specific worker (eg. generic_worker); the master is
# unaffected by this.
systemctl restart matrix-synapse-worker@generic_worker.service

# Add a new worker (assuming all configs are set up already)
systemctl enable matrix-synapse-worker@federation_writer.service
systemctl restart matrix-synapse.target
```

Hardening

Optional: If further hardening is desired, the file `override-hardened.conf` may be copied from `contrib/systemd/override-hardened.conf` in this repository to the location

`/etc/systemd/system/matrix-synapse.service.d/override-hardened.conf` (the directory may have to be created). It enables certain sandboxing features in systemd to further secure the synapse service. You may read the comments to understand what the override file is doing. The same file will need to be copied to `/etc/systemd/system/matrix-synapse-worker@.service.d/override-hardened-worker.conf` (this directory may also have to be created) in order to apply the same hardening options to any worker processes.

Once these files have been copied to their appropriate locations, simply reload systemd's manager config files and restart all Synapse services to apply the hardening options. They

will automatically be applied at every restart as long as the override files are present at the specified locations.

```
systemctl daemon-reload

# Restart services
systemctl restart matrix-synapse.target
```

In order to see their effect, you may run `systemd-analyze security matrix-synapse.service` before and after applying the hardening options to see the changes being applied at a glance.

Administration

This section contains information on managing your Synapse homeserver. This includes:

- Managing users, rooms and media via the Admin API.
- Setting up metrics and monitoring to give you insight into your homeserver's health.
- Configuring structured logging.

How to back up a Synapse homeserver

It is critical to maintain good backups of your server, to guard against hardware failure as well as potential corruption due to bugs or administrator error.

This page documents the things you will need to consider backing up as part of a Synapse installation.

Configuration files

Keep a copy of your configuration file (`homeserver.yaml`), as well as any auxiliary config files it refers to such as the `log_config` file, `app_service_config_files`. Often, all such config files will be kept in a single directory such as `/etc/synapse`, which will make this easier.

Server signing key

Your server has a [signing key](#) which it uses to sign events and outgoing federation requests. It is easiest to back it up with your configuration files, but an alternative is to have Synapse create a new signing key if you have to restore.

If you do decide to replace the signing key, you should add the old *public* key to `old_signing_keys`.

Database

Synapse's support for SQLite is only suitable for testing purposes, so for the purposes of this document, we'll assume you are using [PostgreSQL](#).

A full discussion of backup strategies for PostgreSQL is out of scope for this document; see the [PostgreSQL documentation](#) for detailed information.

Synapse-specific details

- Be very careful not to restore into a database that already has tables present. At best, this will error; at worst, it will lead to subtle database inconsistencies.

- The `e2e_one_time_keys_json` table should **not** be backed up, or if it is backed up, should be `TRUNCATE`d after restoring the database before Synapse is started.

[Background: restoring the database to an older backup can cause used one-time-keys to be re-issued, causing subsequent [message decryption errors](#). Clearing all one-time-keys from the database ensures that this cannot happen, and will prompt clients to generate and upload new one-time-keys.]

Quick and easy database backup and restore

Typically, the easiest solution is to use `pg_dump` to take a copy of the whole database. We recommend `pg_dump`'s custom dump format, as it produces significantly smaller backup files.

```
sudo -u postgres pg_dump -Fc --exclude-table-data e2e_one_time_keys_json synapse
> synapse.dump
```

There is no need to stop Postgres or Synapse while `pg_dump` is running: it will take a consistent snapshot of the database.

To restore, you will need to recreate the database as described in [Using Postgres](#), then load the dump into it with `pg_restore`:

```
sudo -u postgres createdb --encoding=UTF8 --locale=C --template=template0 --
owner=synapse_user synapse
sudo -u postgres pg_restore -d synapse < synapse.dump
```

(If you forgot to exclude `e2e_one_time_keys_json` during `pg_dump`, remember to connect to the new database and `TRUNCATE e2e_one_time_keys_json;` before starting Synapse.)

To reiterate: do **not** restore a dump over an existing database.

Again, if you plan to run your homeserver at any sort of production level, we recommend studying the PostgreSQL documentation on backup options.

Media store

Synapse keeps a copy of media uploaded by users, including avatars and message attachments, in its [Media store](#).

It is a directory on the local disk, containing the following directories:

- `local_content`: this is content uploaded by your local users. As a general rule, you should back this up: it may represent the only copy of those media files anywhere in

the federation, and if they are lost, users will see errors when viewing user or room avatars, and messages with attachments.

- `local_thumbnails`: "thumbnails" of images uploaded by your users. If `dynamic_thumbnails` is enabled, these will be regenerated if they are removed from the disk, and there is therefore no need to back them up.

If `dynamic_thumbnails` is *not* enabled (the default): although this can theoretically be regenerated from `local_content`, there is no tooling to do so. We recommend that these are backed up too.

- `remote_content`: this is a cache of content that was uploaded by a user on another server, and has since been requested by a user on your own server.

Typically there is no need to back up this directory: if a file in this directory is removed, Synapse will attempt to fetch it again from the remote server.

- `remote_thumbnails`: thumbnails of images uploaded by users on other servers. As with `remote_content`, there is normally no need to back this up.
- `url_cache`, `url_cache_thumbnails`: temporary caches of files downloaded by the [URL previews](#) feature. These do not need to be backed up.

The Admin API

Authenticate as a server admin

Many of the API calls in the admin api will require an `access_token` for a server admin. (Note that a server admin is distinct from a room admin.)

An existing user can be marked as a server admin by updating the database directly.

Check your `database settings` in the configuration file, connect to the correct database using either `psql [database name]` (if using PostgreSQL) or `sqlite3 path/to/your/database.db` (if using SQLite) and elevate the user `@foo:bar.com` to administrator.

```
UPDATE users SET admin = 1 WHERE name = '@foo:bar.com' ;
```

A new server admin user can also be created using the `register_new_matrix_user` command. This is a script that is distributed as part of synapse. It is possibly already on your `$PATH` depending on how Synapse was installed.

Finding your user's `access_token` is client-dependent, but will usually be shown in the client's settings.

Making an Admin API request

For security reasons, we [recommend](#) that the Admin API (`/_synapse/admin/...`) should be hidden from public view using a reverse proxy. This means you should typically query the Admin API from a terminal on the machine which runs Synapse.

Once you have your `access_token`, you will need to authenticate each request to an Admin API endpoint by providing the token as either a query parameter or a request header. To add it as a request header in cURL:

```
curl --header "Authorization: Bearer <access_token>"  
<the_rest_of_your_API_request>
```

For example, suppose we want to [query the account](#) of the user `@foo:bar.com`. We need an admin access token (e.g. `syt_AjfVef2_L33JNpafeif_0feKJfeaf0CQpoZk`), and we need to know which port Synapse's `client` listener is listening on (e.g. `8008`). Then we can use the following command to request the account information from the Admin API.

```
curl --header "Authorization: Bearer syt_AjfVef2_L33JNpafeif_0feKJfeaf0CQpoZk" -  
X GET http://127.0.0.1:8008/_synapse/admin/v2/users/@foo:bar.com
```

For more details on access tokens in Matrix, please refer to the complete [matrix spec documentation](#).

Account validity API

Note: This API is disabled when MSC3861 is enabled. See [#15582](#)

This API allows a server administrator to manage the validity of an account. To use it, you must enable the account validity feature (under `account_validity`) in Synapse's configuration.

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

Renew account

This API extends the validity of an account by as much time as configured in the `period` parameter from the `account_validity` configuration.

The API is:

```
POST /_synapse/admin/v1/account_validity/validity
```

with the following body:

```
{  
  "user_id": "<user ID for the account to renew>",  
  "expiration_ts": 0,  
  "enable_renewal_emails": true  
}
```

`expiration_ts` is an optional parameter and overrides the expiration date, which otherwise defaults to now + validity period.

`enable_renewal_emails` is also an optional parameter and enables/disables sending renewal emails to the user. Defaults to true.

The API returns with the new expiration date for this account, as a timestamp in milliseconds since epoch:

```
{  
  "expiration_ts": 0  
}
```

Background Updates API

This API allows a server administrator to manage the background updates being run against the database.

Status

This API gets the current status of the background updates.

The API is:

```
GET /_synapse/admin/v1/background_updates/status
```

Returning:

```
{
  "enabled": true,
  "current_updates": [
    "<db_name>": {
      "name": "<background_update_name>",
      "total_item_count": 50,
      "total_duration_ms": 10000.0,
      "average_items_per_ms": 2.2,
    },
  ],
}
```

`enabled` whether the background updates are enabled or disabled.

`db_name` the database name (usually Synapse is configured with a single database named 'master').

For each update:

`name` the name of the update. `total_item_count` total number of "items" processed (the meaning of 'items' depends on the update in question). `total_duration_ms` how long the background process has been running, not including time spent sleeping.

`average_items_per_ms` how many items are processed per millisecond based on an exponential average.

Enabled

This API allows pausing background updates.

Background updates should *not* be paused for significant periods of time, as this can affect the performance of Synapse.

Note: This won't persist over restarts.

Note: This won't cancel any update query that is currently running. This is usually fine since most queries are short lived, except for `CREATE INDEX` background updates which won't be cancelled once started.

The API is:

```
POST /_synapse/admin/v1/background_updates/enable
```

with the following body:

```
{  
  "enabled": false  
}
```

`enabled` sets whether the background updates are enabled or disabled.

The API returns the `enabled` param.

```
{  
  "enabled": false  
}
```

There is also a `GET` version which returns the `enabled` state.

Run

This API schedules a specific background update to run. The job starts immediately after calling the API.

The API is:

```
POST /_synapse/admin/v1/background_updates/start_job
```

with the following body:

```
{  
  "job_name": "populate_stats_process_rooms"  
}
```

The following JSON body parameters are available:

- `job_name` - A string which job to run. Valid values are:

- `populate_stats_process_rooms` - Recalculate the stats for all rooms.
- `regenerate_directory` - Recalculate the [user directory](#) if it is stale or out of sync.

Show reported events

This API returns information about reported events.

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

The api is:

```
GET /_synapse/admin/v1/event_reports?from=0&limit=10
```

It returns a JSON body like the following:

```
{
  "event_reports": [
    {
      "event_id": "$bNUFCwGzWca1meCGkjp-zwslF-GfVcXukvRLI1_FaVY",
      "id": 2,
      "reason": "foo",
      "score": -100,
      "received_ts": 1570897107409,
      "canonical_alias": "#alias1:matrix.org",
      "room_id": "!ERAgBpSOCuTJqQPk:matrix.org",
      "name": "Matrix HQ",
      "sender": "@foobar:matrix.org",
      "user_id": "@foo:matrix.org"
    },
    {
      "event_id": "$3IcdZsDaN_En-S1DF4EMCy3v4gNRKe0Js8W5qT0Kj4I",
      "id": 3,
      "reason": "bar",
      "score": -100,
      "received_ts": 1598889612059,
      "canonical_alias": "#alias2:matrix.org",
      "room_id": "!eGvUQuTCKHGVwNMOjv:matrix.org",
      "name": "Your room name here",
      "sender": "@foobar:matrix.org",
      "user_id": "@bar:matrix.org"
    }
  ],
  "next_token": 2,
  "total": 4
}
```

To paginate, check for `next_token` and if present, call the endpoint again with `from` set to the value of `next_token`. This will return a new page.

If the endpoint does not return a `next_token` then there are no more reports to paginate through.

URL parameters:

- **limit**: integer - Is optional but is used for pagination, denoting the maximum number of items to return in this call. Defaults to **100**.
- **from**: integer - Is optional but used for pagination, denoting the offset in the returned results. This should be treated as an opaque value and not explicitly set to anything other than the return value of **next_token** from a previous call. Defaults to **0**.
- **dir**: string - Direction of event report order. Whether to fetch the most recent first (**b**) or the oldest first (**f**). Defaults to **b**.
- **user_id**: optional string - Filter by the user ID of the reporter. This is the user who reported the event and wrote the reason.
- **room_id**: optional string - Filter by room id.
- **event_sender_user_id**: optional string - Filter by the sender of the reported event. This is the user who the report was made against.

Response

The following fields are returned in the JSON response body:

- **id**: integer - ID of event report.
- **received_ts**: integer - The timestamp (in milliseconds since the unix epoch) when this report was sent.
- **room_id**: string - The ID of the room in which the event being reported is located.
- **name**: string - The name of the room.
- **event_id**: string - The ID of the reported event.
- **user_id**: string - This is the user who reported the event and wrote the reason.
- **reason**: string - Comment made by the **user_id** in this report. May be blank or **null**.
- **score**: integer - Content is reported based upon a negative score, where -100 is "most offensive" and 0 is "inoffensive". May be **null**.
- **sender**: string - This is the ID of the user who sent the original message/event that was reported.
- **canonical_alias**: string - The canonical alias of the room. **null** if the room does not have a canonical alias set.
- **next_token**: integer - Indication for pagination. See above.
- **total**: integer - Total number of event reports related to the query (**user_id** and **room_id**).

Show details of a specific event report

This API returns information about a specific event report.

The api is:

```
GET /_synapse/admin/v1/event_reports/<report_id>
```

It returns a JSON body like the following:

```
{
  "event_id": "$bNUFCwGzWca1meCGkjp-zws1F-GfVcXukvRLI1_FaVY",
  "event_json": {
    "auth_events": [
      "$YK4arsKKcc0LRoe700pS8DSj0vUT4NDv0HfInlMFw2M",
      "$oggsNXxzPFRE3y53SUND7nsj69-QzKv03a1RucHu-ws"
    ],
    "content": {
      "body": "matrix.org: This Week in Matrix",
      "format": "org.matrix.custom.html",
      "formatted_body": "<strong>matrix.org</strong>:<br><a href=\"https://matrix.org/blog/\"><strong>This Week in Matrix</strong></a>",
      "msgtype": "m.notice"
    },
    "depth": 546,
    "hashes": {
      "sha256": "xK1//xnmvHJI0vbgXlkI8eEqdvoMmihVDJ9J4SNlsAw"
    },
    "origin": "matrix.org",
    "origin_server_ts": 1592291711430,
    "prev_events": [
      "$YK4arsKKcc0LRoe700pS8DSj0vUT4NDv0HfInlMFw2M"
    ],
    "prev_state": [],
    "room_id": "!ERAgBpSOcCCuTJqQPk:matrix.org",
    "sender": "@foobar:matrix.org",
    "signatures": {
      "matrix.org": {
        "ed25519:a_JaEG":
"cs+0UKW/iHx5pEidbWxh0UiNNHwe46Ai9LwNz+Ah16aWDNszVIe2gaAcVZfvNsBhakQTew51tlKmL2ks
      }
    },
    "type": "m.room.message",
    "unsigned": {
      "age_ts": 1592291711430
    }
  },
  "id": <report_id>,
  "reason": "foo",
  "score": -100,
  "received_ts": 1570897107409,
  "canonical_alias": "#alias1:matrix.org",
  "room_id": "!ERAgBpSOcCCuTJqQPk:matrix.org",
  "name": "Matrix HQ",
  "sender": "@foobar:matrix.org",
  "user_id": "@foo:matrix.org"
}
}
```

URL parameters:

- `report_id`: string - The ID of the event report.

Response

The following fields are returned in the JSON response body:

- **id**: integer - ID of event report.
- **received_ts**: integer - The timestamp (in milliseconds since the unix epoch) when this report was sent.
- **room_id**: string - The ID of the room in which the event being reported is located.
- **name**: string - The name of the room.
- **event_id**: string - The ID of the reported event.
- **user_id**: string - This is the user who reported the event and wrote the reason.
- **reason**: string - Comment made by the **user_id** in this report. May be blank.
- **score**: integer - Content is reported based upon a negative score, where -100 is "most offensive" and 0 is "inoffensive".
- **sender**: string - This is the ID of the user who sent the original message/event that was reported.
- **canonical_alias**: string - The canonical alias of the room. **null** if the room does not have a canonical alias set.
- **event_json**: object - Details of the original event that was reported.

Delete a specific event report

This API deletes a specific event report. If the request is successful, the response body will be an empty JSON object.

The api is:

```
DELETE /_synapse/admin/v1/event_reports/<report_id>
```

URL parameters:

- **report_id**: string - The ID of the event report.

Experimental Features API

This API allows a server administrator to enable or disable some experimental features on a per-user basis. The currently supported features are:

- [MSC3881](#): enable remotely toggling push notifications for another client
- [MSC3575](#): enable experimental sliding sync support
- [MSC4222](#): adding `state_after` to sync v2

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

Enabling/Disabling Features

This API allows a server administrator to enable experimental features for a given user. The request must provide a body containing the user id and listing the features to enable/disable in the following format:

```
{  
  "features": {  
    "msc3026":true,  
    "msc3881":true  
  }  
}
```

where true is used to enable the feature, and false is used to disable the feature.

The API is:

```
PUT /_synapse/admin/v1/experimental_features/<user_id>
```

Listing Enabled Features

To list which features are enabled/disabled for a given user send a request to the following API:

```
GET /_synapse/admin/v1/experimental_features/<user_id>
```

It will return a list of possible features and indicate whether they are enabled or disabled for the user like so:

```
{  
  "features": {  
    "msc3026": true,  
    "msc3881": false,  
    "msc3967": false  
  }  
}
```

Querying media

These APIs allow extracting media information from the homeserver.

Details about the format of the `media_id` and storage of the media in the file system are documented under [media repository](#).

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

List all media in a room

This API gets a list of known media in a room. However, it only shows media from unencrypted events or rooms.

The API is:

```
GET /_synapse/admin/v1/room/<room_id>/media
```

The API returns a JSON body like the following:

```
{
  "local": [
    "mxc://localhost/xwvutsrqponmlkjihgfedcba",
    "mxc://localhost/abcdefghijklmnopqrstuvwxyz"
  ],
  "remote": [
    "mxc://matrix.org/xwvutsrqponmlkjihgfedcba",
    "mxc://matrix.org/abcdefghijklmnopqrstuvwxyz"
  ]
}
```

List all media uploaded by a user

Listing all media that has been uploaded by a local user can be achieved through the use of the [List media uploaded by a user](#) Admin API.

Quarantine media

Quarantining media means that it is marked as inaccessible by users. It applies to any local media, and any locally-cached copies of remote media.

The media file itself (and any thumbnails) is not deleted from the server.

Quarantining media by ID

This API quarantines a single piece of local or remote media.

Request:

```
POST /_synapse/admin/v1/media/quarantine/<server_name>/<media_id>
{}

```

Where `server_name` is in the form of `example.org`, and `media_id` is in the form of `abcdefg12345...`.

Response:

```
{}
```

Remove media from quarantine by ID

This API removes a single piece of local or remote media from quarantine.

Request:

```
POST /_synapse/admin/v1/media/unquarantine/<server_name>/<media_id>
{}

```

Where `server_name` is in the form of `example.org`, and `media_id` is in the form of `abcdefg12345...`.

Response:

```
{}
```

Quarantining media in a room

This API quarantines all local and remote media in a room.

Request:

```
POST /_synapse/admin/v1/room/<room_id>/media/quarantine
```

```
{}
```

Where `room_id` is in the form of `!roomid12345:example.org`.

Response:

```
{  
  "num_quarantined": 10  
}
```

The following fields are returned in the JSON response body:

- `num_quarantined`: integer - The number of media items successfully quarantined

Note that there is a legacy endpoint, `POST`

`/_synapse/admin/v1/quarantine_media/<room_id>`, that operates the same. However, it is deprecated and may be removed in a future release.

Quarantining all media of a user

This API quarantines all *local* media that a *local* user has uploaded. That is to say, if you would like to quarantine media uploaded by a user on a remote homeserver, you should instead use one of the other APIs.

Request:

```
POST /_synapse/admin/v1/user/<user_id>/media/quarantine
```

```
{}
```

URL Parameters

- `user_id`: string - User ID in the form of `@bob:example.org`

Response:

```
{  
  "num_quarantined": 10  
}
```

The following fields are returned in the JSON response body:

- `num_quarantined`: integer - The number of media items successfully quarantined

Protecting media from being quarantined

This API protects a single piece of local media from being quarantined using the above APIs. This is useful for sticker packs and other shared media which you do not want to get quarantined, especially when [quarantining media in a room](#).

Request:

```
POST /_synapse/admin/v1/media/protect/<media_id>
{}

```

Where `media_id` is in the form of `abcdefg12345...`.

Response:

```
{}
```

Unprotecting media from being quarantined

This API reverts the protection of a media.

Request:

```
POST /_synapse/admin/v1/media/unprotect/<media_id>
{}

```

Where `media_id` is in the form of `abcdefg12345...`.

Response:

```
{}
```

Delete local media

This API deletes the *local* media from the disk of your own server. This includes any local thumbnails and copies of media downloaded from remote homeservers. This API will not affect media that has been uploaded to external media repositories (e.g <https://github.com/turt2live/matrix-media-repo/>). See also [Purge Remote Media API](#).

Delete a specific local media

Delete a specific `media_id`.

Request:

```
DELETE /_synapse/admin/v1/media/<server_name>/<media_id>
{}
```

URL Parameters

- `server_name`: string - The name of your local server (e.g `matrix.org`)
- `media_id`: string - The ID of the media (e.g `abcdefghijklmнопqrstuvwxyz`)

Response:

```
{
  "deleted_media": [
    "abcdefghijklmнопqrstuvwxyz"
  ],
  "total": 1
}
```

The following fields are returned in the JSON response body:

- `deleted_media`: an array of strings - List of deleted `media_id`
- `total`: integer - Total number of deleted `media_id`

Delete local media by date or size

Request:

```
POST /_synapse/admin/v1/media/delete?before_ts=<before_ts>
{}
```

Deprecated in Synapse v1.78.0: This API is available at the deprecated endpoint:

```
POST /_synapse/admin/v1/media/<server_name>/delete?before_ts=<before_ts>
{}
```

URL Parameters

- `server_name`: string - The name of your local server (e.g `matrix.org`). *Deprecated in Synapse v1.78.0.*

- **before_ts**: string representing a positive integer - Unix timestamp in milliseconds. Files that were last used before this timestamp will be deleted. It is the timestamp of last access, not the timestamp when the file was created.
- **size_gt**: Optional - string representing a positive integer - Size of the media in bytes. Files that are larger will be deleted. Defaults to **0**.
- **keep_profiles**: Optional - string representing a boolean - Switch to also delete files that are still used in image data (e.g user profile, room avatar). If **false** these files will be deleted. Defaults to **true**.

Response:

```
{
  "deleted_media": [
    "abcdefghijklmnopqrstuvwxyz",
    "abcdefghijklmnopqrstuvwxyz"
  ],
  "total": 2
}
```

The following fields are returned in the JSON response body:

- **deleted_media**: an array of strings - List of deleted **media_id**
- **total**: integer - Total number of deleted **media_id**

Delete media uploaded by a user

You can find details of how to delete multiple media uploaded by a user in [User Admin API](#).

Purge Remote Media API

The purge remote media API allows server admins to purge old cached remote media.

The API is:

```
POST /_synapse/admin/v1/purge_media_cache?before_ts=<unix_timestamp_in_ms>
{}
```

URL Parameters

- **before_ts**: string representing a positive integer - Unix timestamp in milliseconds. All cached media that was last accessed before this timestamp will be removed.

Response:

```
{  
  "deleted": 10  
}
```

The following fields are returned in the JSON response body:

- **deleted**: integer - The number of media items successfully deleted

If the user re-requests purged remote media, synapse will re-request the media from the originating server.

Purge History API

The purge history API allows server admins to purge historic events from their database, reclaiming disk space.

Depending on the amount of history being purged a call to the API may take several minutes or longer. During this period users will not be able to paginate further back in the room from the point being purged from.

Note that Synapse requires at least one message in each room, so it will never delete the last message in a room.

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

The API is:

```
POST /_synapse/admin/v1/purge_history/<room_id>[/<event_id>]
```

By default, events sent by local users are not deleted, as they may represent the only copies of this content in existence. (Events sent by remote users are deleted.)

Room state data (such as joins, leaves, topic) is always preserved.

To delete local message events as well, set `delete_local_events` in the body:

```
{  
  "delete_local_events": true  
}
```

The caller must specify the point in the room to purge up to. This can be specified by including an `event_id` in the URL, or by setting a `purge_up_to_event_id` or `purge_up_to_ts` in the request body. If an event id is given, that event (and others at the same graph depth) will be retained. If `purge_up_to_ts` is given, it should be a timestamp since the unix epoch, in milliseconds.

The API starts the purge running, and returns immediately with a JSON body with a purge id:

```
{  
  "purge_id": "<opaque_id>"  
}
```

Purge status query

It is possible to poll for updates on recent purges with a second API;

```
GET /_synapse/admin/v1/purge_history_status/<purge_id>
```

This API returns a JSON body like the following:

```
{  
  "status": "active"  
}
```

The status will be one of `active`, `complete`, or `failed`.

If `status` is `failed` there will be a string `error` with the error message.

Reclaim disk space (Postgres)

To reclaim the disk space and return it to the operating system, you need to run `VACUUM FULL;` on the database.

<https://www.postgresql.org/docs/current/sql-vacuum.html>

Shared-Secret Registration

Note: This API is disabled when MSC3861 is enabled. See [#15582](#)

This API allows for the creation of users in an administrative and non-interactive way. This is generally used for bootstrapping a Synapse instance with administrator accounts.

To authenticate yourself to the server, you will need both the shared secret (`registration_shared_secret` in the homeserver configuration), and a one-time nonce. If the registration shared secret is not configured, this API is not enabled.

To fetch the nonce, you need to request one from the API:

```
> GET /_synapse/admin/v1/register
< {"nonce": "thisisanonce"}
```

Once you have the nonce, you can make a `POST` to the same URL with a JSON body containing the nonce, username, password, whether they are an admin (optional, `False` by default), and a HMAC digest of the content. Also you can set the displayname (optional, `username` by default).

As an example:

```
> POST /_synapse/admin/v1/register
> {
  "nonce": "thisisanonce",
  "username": "pepper_roni",
  "displayname": "Pepper Roni",
  "password": "pizza",
  "admin": true,
  "mac": "mac_digest_here"
}

< {
  "access_token": "token_here",
  "user_id": "@pepper_roni:localhost",
  "home_server": "test",
  "device_id": "device_id_here"
}
```

The MAC is the hex digest output of the HMAC-SHA1 algorithm, with the key being the shared secret and the content being the nonce, user, password, either the string "admin" or "notadmin", and optionally the user_type each separated by NULs.

Here is an easy way to generate the HMAC digest if you have Bash and OpenSSL:

```
# Update these values and then paste this code block into a bash terminal
nonce='thisisanonce'
username='pepper_roni'
password='pizza'
admin='admin'
secret='shared_secret'

printf '%s\0%s\0%s\0%s' "$nonce" "$username" "$password" "$admin" |
openssl sha1 -hmac "$secret" |
awk '{print $2}'
```

For an example of generation in Python:

```
import hmac, hashlib

def generate_mac(nonce, user, password, admin=False, user_type=None):

    mac = hmac.new(
        key=shared_secret,
        digestmod=hashlib.sha1,
    )

    mac.update(nonce.encode('utf8'))
    mac.update(b"\x00")
    mac.update(user.encode('utf8'))
    mac.update(b"\x00")
    mac.update(password.encode('utf8'))
    mac.update(b"\x00")
    mac.update(b"admin" if admin else b"notadmin")
    if user_type:
        mac.update(b"\x00")
        mac.update(user_type.encode('utf8'))

    return mac.hexdigest()
```

Registration Tokens

Note: This API is disabled when MSC3861 is enabled. See [#15582](#)

This API allows you to manage tokens which can be used to authenticate registration requests, as proposed in [MSC3231](#) and stabilised in version 1.2 of the Matrix specification. To use it, you will need to enable the `registration_requires_token` config option, and authenticate by providing an `access_token` for a server admin: see [Admin API](#).

Registration token objects

Most endpoints make use of JSON objects that contain details about tokens. These objects have the following fields:

- `token`: The token which can be used to authenticate registration.
- `uses_allowed`: The number of times the token can be used to complete a registration before it becomes invalid.
- `pending`: The number of pending uses the token has. When someone uses the token to authenticate themselves, the pending counter is incremented so that the token is not used more than the permitted number of times. When the person completes registration the pending counter is decremented, and the completed counter is incremented.
- `completed`: The number of times the token has been used to successfully complete a registration.
- `expiry_time`: The latest time the token is valid. Given as the number of milliseconds since 1970-01-01 00:00:00 UTC (the start of the Unix epoch). To convert this into a human-readable form you can remove the milliseconds and use the `date` command. For example, `date -d '@1625394937'`.

List all tokens

Lists all tokens and details about them. If the request is successful, the top level JSON object will have a `registration_tokens` key which is an array of registration token objects.

```
GET /_synapse/admin/v1/registration_tokens
```

Optional query parameters:

- `valid`: `true` or `false`. If `true`, only valid tokens are returned. If `false`, only tokens that have expired or have had all uses exhausted are returned. If omitted, all tokens

are returned regardless of validity.

Example:

```
GET /_synapse/admin/v1/registration_tokens
```

200 OK

```
{
  "registration_tokens": [
    {
      "token": "abcd",
      "uses_allowed": 3,
      "pending": 0,
      "completed": 1,
      "expiry_time": null
    },
    {
      "token": "pqrs",
      "uses_allowed": 2,
      "pending": 1,
      "completed": 1,
      "expiry_time": null
    },
    {
      "token": "wxyz",
      "uses_allowed": null,
      "pending": 0,
      "completed": 9,
      "expiry_time": 1625394937000      // 2021-07-04 10:35:37 UTC
    }
  ]
}
```

Example using the `valid` query parameter:

```
GET /_synapse/admin/v1/registration_tokens?valid=false
```

200 OK

```
{  
  "registration_tokens": [  
    {  
      "token": "pqrs",  
      "uses_allowed": 2,  
      "pending": 1,  
      "completed": 1,  
      "expiry_time": null  
    },  
    {  
      "token": "wxyz",  
      "uses_allowed": null,  
      "pending": 0,  
      "completed": 9,  
      "expiry_time": 1625394937000 // 2021-07-04 10:35:37 UTC  
    }  
  ]  
}
```

Get one token

Get details about a single token. If the request is successful, the response body will be a registration token object.

```
GET /_synapse/admin/v1/registration_tokens/<token>
```

Path parameters:

- **token**: The registration token to return details of.

Example:

```
GET /_synapse/admin/v1/registration_tokens/abcd
```

200 OK

```
{  
  "token": "abcd",  
  "uses_allowed": 3,  
  "pending": 0,  
  "completed": 1,  
  "expiry_time": null  
}
```

Create token

Create a new registration token. If the request is successful, the newly created token will be returned as a registration token object in the response body.

```
POST /_synapse/admin/v1/registration_tokens/new
```

The request body must be a JSON object and can contain the following fields:

- **token**: The registration token. A string of no more than 64 characters that consists only of characters matched by the regex `[A-Za-z0-9._~-]`. Default: randomly generated.
- **uses_allowed**: The integer number of times the token can be used to complete a registration before it becomes invalid. Default: `null` (unlimited uses).
- **expiry_time**: The latest time the token is valid. Given as the number of milliseconds since 1970-01-01 00:00:00 UTC (the start of the Unix epoch). You could use, for example, `date '+%s000' -d 'tomorrow'`. Default: `null` (token does not expire).
- **length**: The length of the token randomly generated if `token` is not specified. Must be between 1 and 64 inclusive. Default: `16`.

If a field is omitted the default is used.

Example using defaults:

```
POST /_synapse/admin/v1/registration_tokens/new
```

```
{}
```

```
200 OK
```

```
{
  "token": "0M-9jbkf2t_Tgiw1",
  "uses_allowed": null,
  "pending": 0,
  "completed": 0,
  "expiry_time": null
}
```

Example specifying some fields:

```
POST /_synapse/admin/v1/registration_tokens/new
```

```
{
  "token": "defg",
  "uses_allowed": 1
}
```

```
200 OK
```

```
{  
  "token": "defg",  
  "uses_allowed": 1,  
  "pending": 0,  
  "completed": 0,  
  "expiry_time": null  
}
```

Update token

Update the number of allowed uses or expiry time of a token. If the request is successful, the updated token will be returned as a registration token object in the response body.

```
PUT /_synapse/admin/v1/registration_tokens/<token>
```

Path parameters:

- **token**: The registration token to update.

The request body must be a JSON object and can contain the following fields:

- **uses_allowed**: The integer number of times the token can be used to complete a registration before it becomes invalid. By setting **uses_allowed** to **0** the token can be easily made invalid without deleting it. If **null** the token will have an unlimited number of uses.
- **expiry_time**: The latest time the token is valid. Given as the number of milliseconds since 1970-01-01 00:00:00 UTC (the start of the Unix epoch). If **null** the token will not expire.

If a field is omitted its value is not modified.

Example:

```
PUT /_synapse/admin/v1/registration_tokens/defg  
{  
  "expiry_time": 4781243146000    // 2121-07-06 11:05:46 UTC  
}
```

200 OK

```
{  
  "token": "defg",  
  "uses_allowed": 1,  
  "pending": 0,  
  "completed": 0,  
  "expiry_time": 4781243146000  
}
```

Delete token

Delete a registration token. If the request is successful, the response body will be an empty JSON object.

DELETE /_synapse/admin/v1/registration_tokens/<token>

Path parameters:

- **token**: The registration token to delete.

Example:

DELETE /_synapse/admin/v1/registration_tokens/wxyz**200 OK**

```
{}
```

Errors

If a request fails a "standard error response" will be returned as defined in the [Matrix Client-Server API specification](#).

For example, if the token specified in a path parameter does not exist a **404 Not Found** error will be returned.

GET /_synapse/admin/v1/registration_tokens/1234**404 Not Found**

```
{  
  "errcode": "M_NOT_FOUND",  
  "error": "No such registration token: 1234"  
}
```

Edit Room Membership API

This API allows an administrator to join a user account with a given `user_id` to a room with a given `room_id_or_alias`. You can only modify the membership of local users. The server administrator must be in the room and have permission to invite users.

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

Parameters

The following parameters are available:

- `user_id` - Fully qualified user: for example, `@user:server.com`.
- `room_id_or_alias` - The room identifier or alias to join: for example, `!636q39766251:server.com`.

Usage

```
POST /_synapse/admin/v1/join/<room_id_or_alias>
{
  "user_id": "@user:server.com"
}
```

Response:

```
{
  "room_id": "!636q39766251:server.com"
}
```

List Room API

The List Room admin API allows server admins to get a list of rooms on their server. There are various parameters available that allow for filtering and sorting the returned list. This API supports pagination.

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

Parameters

The following query parameters are available:

- `from` - Offset in the returned list. Defaults to `0`.
- `limit` - Maximum amount of rooms to return. Defaults to `100`.
- `order_by` - The method in which to sort the returned list of rooms. Valid values are:
 - `alphabetical` - Same as `name`. This is deprecated.
 - `size` - Same as `joined_members`. This is deprecated.
 - `name` - Rooms are ordered alphabetically by room name. This is the default.
 - `canonical_alias` - Rooms are ordered alphabetically by main alias address of the room.
 - `joined_members` - Rooms are ordered by the number of members. Largest to smallest.
 - `joined_local_members` - Rooms are ordered by the number of local members. Largest to smallest.
 - `version` - Rooms are ordered by room version. Largest to smallest.
 - `creator` - Rooms are ordered alphabetically by creator of the room.
 - `encryption` - Rooms are ordered alphabetically by the end-to-end encryption algorithm.
 - `federatable` - Rooms are ordered by whether the room is federatable.
 - `public` - Rooms are ordered by visibility in room list.
 - `join_rules` - Rooms are ordered alphabetically by join rules of the room.
 - `guest_access` - Rooms are ordered alphabetically by guest access option of the room.
 - `history_visibility` - Rooms are ordered alphabetically by visibility of history of the room.
 - `state_events` - Rooms are ordered by number of state events. Largest to smallest.
- `dir` - Direction of room order. Either `f` for forwards or `b` for backwards. Setting this value to `b` will reverse the above sort order. Defaults to `f`.

- **search_term** - Filter rooms by their room name, canonical alias and room id. Specifically, rooms are selected if the search term is contained in
 - the room's name,
 - the local part of the room's canonical alias, or
 - the complete (local and server part) room's id (case sensitive).
- **public_rooms** - Optional flag to filter public rooms. If **true**, only public rooms are queried. If **false**, public rooms are excluded from the query. When the flag is absent (the default), **both** public and non-public rooms are included in the search results.
- **empty_rooms** - Optional flag to filter empty rooms. A room is empty if joined_members is zero. If **true**, only empty rooms are queried. If **false**, empty rooms are excluded from the query. When the flag is absent (the default), **both** empty and non-empty rooms are included in the search results.

Defaults to no filtering.

Response

The following fields are possible in the JSON response body:

- **rooms** - An array of objects, each containing information about a room.
 - Room objects contain the following fields:
 - **room_id** - The ID of the room.
 - **name** - The name of the room.
 - **canonical_alias** - The canonical (main) alias address of the room.
 - **joined_members** - How many users are currently in the room.
 - **joined_local_members** - How many local users are currently in the room.
 - **version** - The version of the room as a string.
 - **creator** - The **user_id** of the room creator.
 - **encryption** - Algorithm of end-to-end encryption of messages. Is **null** if encryption is not active.
 - **federatable** - Whether users on other servers can join this room.
 - **public** - Whether the room is visible in room directory.
 - **join_rules** - The type of rules used for users wishing to join this room. One of: ["public", "knock", "invite", "private"].
 - **guest_access** - Whether guests can join the room. One of: ["can_join", "forbidden"].
 - **history_visibility** - Who can see the room history. One of: ["invited", "joined", "shared", "world_readable"].
 - **state_events** - Total number of state_events of a room. Complexity of the room.
 - **room_type** - The type of the room taken from the room's creation event; for example "m.space" if the room is a space. If the room does not define a type, the value will be **null**.

- `offset` - The current pagination offset in rooms. This parameter should be used instead of `next_token` for room offset as `next_token` is not intended to be parsed.
- `total_rooms` - The total number of rooms this query can return. Using this and `offset`, you have enough information to know the current progression through the list.
- `next_batch` - If this field is present, we know that there are potentially more rooms on the server that did not all fit into this response. We can use `next_batch` to get the "next page" of results. To do so, simply repeat your request, setting the `from` parameter to the value of `next_batch`.
- `prev_batch` - If this field is present, it is possible to paginate backwards. Use `prev_batch` for the `from` value in the next request to get the "previous page" of results.

The API is:

A standard request with no filtering:

```
GET /_synapse/admin/v1/rooms
```

A response body like the following is returned:

```
{
  "rooms": [
    {
      "room_id": "!0GEhHVWSdvArJzumhm:matrix.org",
      "name": "Matrix HQ",
      "canonical_alias": "#matrix:matrix.org",
      "joined_members": 8326,
      "joined_local_members": 2,
      "version": "1",
      "creator": "@foo:matrix.org",
      "encryption": null,
      "federatable": true,
      "public": true,
      "join_rules": "invite",
      "guest_access": null,
      "history_visibility": "shared",
      "state_events": 93534,
      "room_type": "m.space"
    },
    ... (8 hidden items) ...
    {
      "room_id": "!xYvNcQPhnkrdUmYczI:matrix.org",
      "name": "This Week In Matrix (TWIM)",
      "canonical_alias": "#twim:matrix.org",
      "joined_members": 314,
      "joined_local_members": 20,
      "version": "4",
      "creator": "@foo:matrix.org",
      "encryption": "m.megolm.v1.aes-sha2",
      "federatable": true,
      "public": false,
      "join_rules": "invite",
      "guest_access": null,
      "history_visibility": "shared",
      "state_events": 8345,
      "room_type": null
    }
  ],
  "offset": 0,
  "total_rooms": 10
}
```

Filtering by room name:

```
GET /_synapse/admin/v1/rooms?search_term=TWIM
```

A response body like the following is returned:

```
{  
  "rooms": [  
    {  
      "room_id": "!xYvNcQPhnkrdUmYczI:matrix.org",  
      "name": "This Week In Matrix (TWIM)",  
      "canonical_alias": "#twim:matrix.org",  
      "joined_members": 314,  
      "joined_local_members": 20,  
      "version": "4",  
      "creator": "@foo:matrix.org",  
      "encryption": "m.megolm.v1.aes-sha2",  
      "federatable": true,  
      "public": false,  
      "join_rules": "invite",  
      "guest_access": null,  
      "history_visibility": "shared",  
      "state_events": 8,  
      "room_type": null  
    }  
  ],  
  "offset": 0,  
  "total_rooms": 1  
}
```

Paginating through a list of rooms:

```
GET /_synapse/admin/v1/rooms?order_by=size
```

A response body like the following is returned:

```
{
  "rooms": [
    {
      "room_id": "!0GEhHVWSdvArJzumhm:matrix.org",
      "name": "Matrix HQ",
      "canonical_alias": "#matrix:matrix.org",
      "joined_members": 8326,
      "joined_local_members": 2,
      "version": "1",
      "creator": "@foo:matrix.org",
      "encryption": null,
      "federatable": true,
      "public": true,
      "join_rules": "invite",
      "guest_access": null,
      "history_visibility": "shared",
      "state_events": 93534,
      "room_type": null
    },
    ... (98 hidden items) ...
    {
      "room_id": "!xYvNcQPhnkrdUmYczI:matrix.org",
      "name": "This Week In Matrix (TWIM)",
      "canonical_alias": "#twim:matrix.org",
      "joined_members": 314,
      "joined_local_members": 20,
      "version": "4",
      "creator": "@foo:matrix.org",
      "encryption": "m.megolm.v1.aes-sha2",
      "federatable": true,
      "public": false,
      "join_rules": "invite",
      "guest_access": null,
      "history_visibility": "shared",
      "state_events": 8345,
      "room_type": "m.space"
    }
  ],
  "offset": 0,
  "total_rooms": 150,
  "next_token": 100
}
```

The presence of the `next_token` parameter tells us that there are more rooms than returned in this request, and we need to make another request to get them. To get the next batch of room results, we repeat our request, setting the `from` parameter to the value of `next_token`.

```
GET /_synapse/admin/v1/rooms?order_by=size&from=100
```

A response body like the following is returned:

```
{
  "rooms": [
    {
      "room_id": "!mscvqgqpHYjBGDxNym:matrix.org",
      "name": "Music Theory",
      "canonical_alias": "#musictheory:matrix.org",
      "joined_members": 127,
      "joined_local_members": 2,
      "version": "1",
      "creator": "@foo:matrix.org",
      "encryption": null,
      "federatable": true,
      "public": true,
      "join_rules": "invite",
      "guest_access": null,
      "history_visibility": "shared",
      "state_events": 93534,
      "room_type": "m.space"
    },
    ... (48 hidden items) ...
  ],
  "room_id": "!twcBhHVdZlQWuuxBhN:termina.org.uk",
  "name": "weechat-matrix",
  "canonical_alias": "#weechat-matrix:termina.org.uk",
  "joined_members": 137,
  "joined_local_members": 20,
  "version": "4",
  "creator": "@foo:termina.org.uk",
  "encryption": null,
  "federatable": true,
  "public": true,
  "join_rules": "invite",
  "guest_access": null,
  "history_visibility": "shared",
  "state_events": 8345,
  "room_type": null
  }
],
"offset": 100,
"prev_batch": 0,
"total_rooms": 150
}
```

Once the `next_token` parameter is no longer present, we know we've reached the end of the list.

Room Details API

The Room Details admin API allows server admins to get all details of a room.

The following fields are possible in the JSON response body:

- `room_id` - The ID of the room.
- `name` - The name of the room.
- `topic` - The topic of the room.
- `avatar` - The `mxc` URI to the avatar of the room.
- `canonical_alias` - The canonical (main) alias address of the room.
- `joined_members` - How many users are currently in the room.
- `joined_local_members` - How many local users are currently in the room.
- `joined_local_devices` - How many local devices are currently in the room.
- `version` - The version of the room as a string.
- `creator` - The `user_id` of the room creator.
- `encryption` - Algorithm of end-to-end encryption of messages. Is `null` if encryption is not active.
- `federatable` - Whether users on other servers can join this room.
- `public` - Whether the room is visible in room directory.
- `join_rules` - The type of rules used for users wishing to join this room. One of: `["public", "knock", "invite", "private"]`.
- `guest_access` - Whether guests can join the room. One of: `["can_join", "forbidden"]`.
- `history_visibility` - Who can see the room history. One of: `["invited", "joined", "shared", "world_readable"]`.
- `state_events` - Total number of state_events of a room. Complexity of the room.
- `room_type` - The type of the room taken from the room's creation event; for example `"m.space"` if the room is a space. If the room does not define a type, the value will be `null`.
- `forgotten` - Whether all local users have `forgotten` the room.

The API is:

```
GET /_synapse/admin/v1/rooms/<room_id>
```

A response body like the following is returned:

```
{
  "room_id": "!mscvqgqpHYjBGDxNym:matrix.org",
  "name": "Music Theory",
  "avatar": "mxc://matrix.org/AQDaVFlbkQoErd0gqWRgiGSV",
  "topic": "Theory, Composition, Notation, Analysis",
  "canonical_alias": "#musictheory:matrix.org",
  "joined_members": 127,
  "joined_local_members": 2,
  "joined_local_devices": 2,
  "version": "1",
  "creator": "@foo:matrix.org",
  "encryption": null,
  "federatable": true,
  "public": true,
  "join_rules": "invite",
  "guest_access": null,
  "history_visibility": "shared",
  "state_events": 93534,
  "room_type": "m.space",
  "forgotten": false
}
```

Changed in Synapse 1.66: Added the `forgotten` key to the response body.

Room Members API

The Room Members admin API allows server admins to get a list of all members of a room.

The response includes the following fields:

- `members` - A list of all the members that are present in the room, represented by their ids.
- `total` - Total number of members in the room.

The API is:

```
GET /_synapse/admin/v1/rooms/<room_id>/members
```

A response body like the following is returned:

```
{
  "members": [
    "@foo:matrix.org",
    "@bar:matrix.org",
    "@foobar:matrix.org"
  ],
  "total": 3
}
```

Room State API

The Room State admin API allows server admins to get a list of all state events in a room.

The response includes the following fields:

- **state** - The current state of the room at the time of request.

The API is:

```
GET /_synapse/admin/v1/rooms/<room_id>/state
```

Parameters

The following query parameter is available:

- **type** - The type of room state event to filter by, eg "m.room.create". If provided, only state events of this type will be returned (regardless of their **state_key** value).

A response body like the following is returned:

```
{
  "state": [
    {"type": "m.room.create", "state_key": "", "etc": true},
    {"type": "m.room.power_levels", "state_key": "", "etc": true},
    {"type": "m.room.name", "state_key": "", "etc": true}
  ]
}
```

Room Messages API

The Room Messages admin API allows server admins to get all messages sent to a room in a given timeframe. There are various parameters available that allow for filtering and ordering the returned list. This API supports pagination.

To use it, you will need to authenticate by providing an **access_token** for a server admin: see [Admin API](#).

This endpoint mirrors the [Matrix Spec defined Messages API](#).

The API is:

```
GET /_synapse/admin/v1/rooms/<room_id>/messages
```

Parameters

The following path parameters are required:

- **room_id** - The ID of the room you wish to fetch messages from.

The following query parameters are available:

- **from** (required) - The token to start returning events from. This token can be obtained from a prev_batch or next_batch token returned by the /sync endpoint, or from an end token returned by a previous request to this endpoint.
- **to** - The token to stop returning events at.
- **limit** - The maximum number of events to return. Defaults to **10**.
- **filter** - A JSON RoomEventFilter to filter returned events with.
- **dir** - The direction to return events from. Either **f** for forwards or **b** for backwards. Setting this value to **b** will reverse the above sort order. Defaults to **f**.

Response

The following fields are possible in the JSON response body:

- **chunk** - A list of room events. The order depends on the dir parameter. Note that an empty chunk does not necessarily imply that no more events are available. Clients should continue to paginate until no end property is returned.
- **end** - A token corresponding to the end of chunk. This token can be passed back to this endpoint to request further events. If no further events are available, this property is omitted from the response.
- **start** - A token corresponding to the start of chunk.
- **state** - A list of state events relevant to showing the chunk.

Example

For more details on each chunk, read the [Matrix specification](#).

```
{  
  "chunk": [  
    {  
      "content": {  
        "body": "This is an example text message",  
        "format": "org.matrix.custom.html",  
        "formatted_body": "<b>This is an example text message</b>",  
        "msgtype": "m.text"  
      },  
      "event_id": "$143273582443PhrSn@example.org",  
      "origin_server_ts": 1432735824653,  
      "room_id": "!636q39766251@example.com",  
      "sender": "@example@example.org",  
      "type": "m.room.message",  
      "unsigned": {  
        "age": 1234  
      }  
    },  
    {  
      "content": {  
        "name": "The room name"  
      },  
      "event_id": "$143273582443PhrSn@example.org",  
      "origin_server_ts": 1432735824653,  
      "room_id": "!636q39766251@example.com",  
      "sender": "@example@example.org",  
      "state_key": "",  
      "type": "m.room.name",  
      "unsigned": {  
        "age": 1234  
      }  
    },  
    {  
      "content": {  
        "body": "Gangnam Style",  
        "info": {  
          "duration": 2140786,  
          "h": 320,  
          "mimetype": "video/mp4",  
          "size": 1563685,  
          "thumbnail_info": {  
            "h": 300,  
            "mimetype": "image/jpeg",  
            "size": 46144,  
            "w": 300  
          },  
          "thumbnail_url": "mxc://example.org/FHyPlCeYUSFFxlgbQYZmoEoe",  
          "w": 480  
        },  
        "msgtype": "m.video",  
        "url": "mxc://example.org/a526eYUSFFxlgbQYZmo442"  
      },  
      "event_id": "$143273582443PhrSn@example.org",  
      "origin_server_ts": 1432735824653,  
      "room_id": "!636q39766251@example.com",  
      "sender": "@example@example.org",  
      "type": "m.room.message",  
    }  
  ]  
}
```

```

        "unsigned": {
            "age": 1234
        }
    },
    "end": "t47409-4357353_219380_26003_2265",
    "start": "t47429-4392820_219380_26003_2265"
}

```

Room Timestamp to Event API

The Room Timestamp to Event API endpoint fetches the `event_id` of the closest event to the given timestamp (`ts` query parameter) in the given direction (`dir` query parameter).

Useful for cases like jump to date so you can start paginating messages from a given date in the archive.

The API is:

```
GET /_synapse/admin/v1/rooms/<room_id>/timestamp_to_event
```

Parameters

The following path parameters are required:

- `room_id` - The ID of the room you wish to check.

The following query parameters are available:

- `ts` - a timestamp in milliseconds where we will find the closest event in the given direction.
- `dir` - can be `f` or `b` to indicate forwards and backwards in time from the given timestamp. Defaults to `f`.

Response

- `event_id` - The event ID closest to the given timestamp.
- `origin_server_ts` - The timestamp of the event in milliseconds since the Unix epoch.

Block Room API

The Block Room admin API allows server admins to block and unblock rooms, and query to see if a given room is blocked. This API can be used to pre-emptively block a room, even if it's unknown to this homeserver. Users will be prevented from joining a blocked room.

Block or unblock a room

The API is:

```
PUT /_synapse/admin/v1/rooms/<room_id>/block
```

with a body of:

```
{  
  "block": true  
}
```

A response body like the following is returned:

```
{  
  "block": true  
}
```

Parameters

The following parameters should be set in the URL:

- `room_id` - The ID of the room.

The following JSON body parameters are available:

- `block` - If `true` the room will be blocked and if `false` the room will be unblocked.

Response

The following fields are possible in the JSON response body:

- `block` - A boolean. `true` if the room is blocked, otherwise `false`

Get block status

The API is:

```
GET /_synapse/admin/v1/rooms/<room_id>/block
```

A response body like the following is returned:

```
{  
  "block": true,  
  "user_id": "<user_id>"  
}
```

Parameters

The following parameters should be set in the URL:

- `room_id` - The ID of the room.

Response

The following fields are possible in the JSON response body:

- `block` - A boolean. `true` if the room is blocked, otherwise `false`
- `user_id` - An optional string. If the room is blocked (`block` is `true`) shows the user who has added the room to blocking list. Otherwise it is not displayed.

Delete Room API

The Delete Room admin API allows server admins to remove rooms from the server and block these rooms.

Shuts down a room. Moves all local users and room aliases automatically to a new room if `new_room_user_id` is set. Otherwise local users only leave the room without any information.

The new room will be created with the user specified by the `new_room_user_id` parameter as room administrator and will contain a message explaining what happened. Users invited to the new room will have power level `-10` by default, and thus be unable to speak.

If `block` is `true`, users will be prevented from joining the old room. This option can in [Version 1](#) also be used to pre-emptively block a room, even if it's unknown to this homeserver. In this case, the room will be blocked, and no further action will be taken. If `block` is `false`, attempting to delete an unknown room is invalid and will be rejected as a bad request.

This API will remove all trace of the old room from your database after removing all local users. If `purge` is `true` (the default), all traces of the old room will be removed from your database after removing all local users. If you do not want this to happen, set `purge` to `false`. Depending on the amount of history being purged, a call to the API may take several minutes or longer.

The local server will only have the power to move local user and room aliases to the new room. Users on other servers will be unaffected.

Version 1 (old version)

This version works synchronously. That means you only get the response once the server has finished the action, which may take a long time. If you request the same action a second time, and the server has not finished the first one, the second request will block. This is fixed in version 2 of this API. The parameters are the same in both APIs. This API will become deprecated in the future.

The API is:

```
DELETE /_synapse/admin/v1/rooms/<room_id>
```

with a body of:

```
{
  "new_room_user_id": "@someuser@example.com",
  "room_name": "Content Violation Notification",
  "message": "Bad Room has been shutdown due to content violations on this
server. Please review our Terms of Service.",
  "block": true,
  "purge": true
}
```

A response body like the following is returned:

```
{
  "kicked_users": [
    "@foobar@example.com"
  ],
  "failed_to_kick_users": [],
  "local_aliases": [
    "#badroom@example.com",
    "#evilsaloon@example.com"
  ],
  "new_room_id": "!newroomid@example.com"
}
```

The parameters and response values have the same format as [version 2](#) of the API.

Version 2 (new version)

Note: This API is new, experimental and "subject to change".

This version works asynchronously, meaning you get the response from server immediately while the server works on that task in background. You can then request the status of the action to check if it has completed.

The API is:

```
DELETE /_synapse/admin/v2/rooms/<room_id>
```

with a body of:

```
{
  "new_room_user_id": "@someuser@example.com",
  "room_name": "Content Violation Notification",
  "message": "Bad Room has been shutdown due to content violations on this
server. Please review our Terms of Service.",
  "block": true,
  "purge": true
}
```

The API starts the shut down and purge running, and returns immediately with a JSON body with a purge id:

```
{
  "delete_id": "<opaque id>"
}
```

Parameters

The following parameters should be set in the URL:

- `room_id` - The ID of the room.

The following JSON body parameters are available:

- `new_room_user_id` - Optional. If set, a new room will be created with this user ID as the creator and admin, and all users in the old room will be moved into that room. If not set, no new room will be created and the users will just be removed from the old room. The user ID must be on the local server, but does not necessarily have to belong to a registered user.
- `room_name` - Optional. A string representing the name of the room that new users will be invited to. Defaults to `Content Violation Notification`
- `message` - Optional. A string containing the first message that will be sent as `new_room_user_id` in the new room. Ideally this will clearly convey why the original room was shut down. Defaults to `Sharing illegal content on this server is not permitted and rooms in violation will be blocked.`
- `block` - Optional. If set to `true`, this room will be added to a blocking list, preventing future attempts to join the room. Rooms can be blocked even if they're not yet known to the homeserver (only with [Version 1](#) of the API). Defaults to `false`.
- `purge` - Optional. If set to `true`, it will remove all traces of the room from your database. Defaults to `true`.
- `force_purge` - Optional, and ignored unless `purge` is `true`. If set to `true`, it will force a purge to go ahead even if there are local users still in the room. Do not use this unless a regular `purge` operation fails, as it could leave those users' clients in a confused state.

The JSON body must not be empty. The body must be at least `{}`.

Status of deleting rooms

Note: This API is new, experimental and "subject to change".

It is possible to query the status of the background task for deleting rooms. The status can be queried up to 24 hours after completion of the task, or until Synapse is restarted (whichever happens first).

Query by `room_id`

With this API you can get the status of all active deletion tasks, and all those completed in the last 24h, for the given `room_id`.

The API is:

```
GET /_synapse/admin/v2/rooms/<room_id>/delete_status
```

A response body like the following is returned:

```
{
  "results": [
    {
      "delete_id": "delete_id1",
      "status": "failed",
      "error": "error message",
      "shutdown_room": {
        "kicked_users": [],
        "failed_to_kick_users": [],
        "local_aliases": [],
        "new_room_id": null
      }
    },
    {
      "delete_id": "delete_id2",
      "status": "purging",
      "shutdown_room": {
        "kicked_users": [
          "@foobar@example.com"
        ],
        "failed_to_kick_users": [],
        "local_aliases": [
          "#badroom@example.com",
          "#evilsaloon@example.com"
        ],
        "new_room_id": "!newroomid@example.com"
      }
    }
  ]
}
```

Parameters

The following parameters should be set in the URL:

- `room_id` - The ID of the room.

Query by `delete_id`

With this API you can get the status of one specific task by `delete_id`.

The API is:

```
GET /_synapse/admin/v2/rooms/delete_status/<delete_id>
```

A response body like the following is returned:

```
{
  "status": "purgging",
  "shutdown_room": {
    "kicked_users": [
      "@foobar@example.com"
    ],
    "failed_to_kick_users": [],
    "local_aliases": [
      "#badroom@example.com",
      "#evilsaloon@example.com"
    ],
    "new_room_id": "!newroomid@example.com"
  }
}
```

Parameters

The following parameters should be set in the URL:

- `delete_id` - The ID for this delete.

Response

The following fields are returned in the JSON response body:

- `results` - An array of objects, each containing information about one task. This field is omitted from the result when you query by `delete_id`. Task objects contain the following fields:
 - `delete_id` - The ID for this purge if you query by `room_id`.
 - `status` - The status will be one of:
 - `shutting_down` - The process is removing users from the room.
 - `purgging` - The process is purging the room and event data from database.
 - `complete` - The process has completed successfully.
 - `failed` - The process is aborted, an error has occurred.
 - `error` - A string that shows an error message if `status` is `failed`. Otherwise this field is hidden.
 - `shutdown_room` - An object containing information about the result of shutting down the room. *Note:* The result is shown after removing the room members. The delete process can still be running. Please pay attention to the `status`.
 - `kicked_users` - An array of users (`user_id`) that were kicked.
 - `failed_to_kick_users` - An array of users (`user_id`) that were not kicked.
 - `local_aliases` - An array of strings representing the local aliases that were migrated from the old room to the new.
 - `new_room_id` - A string representing the room ID of the new room, or `null` if no such room was created.

Undoing room deletions

Note: This guide may be outdated by the time you read it. By nature of room deletions being performed at the database level, the structure can and does change without notice.

First, it's important to understand that a room deletion is very destructive. Undoing a deletion is not as simple as pretending it never happened - work has to be done to move forward instead of resetting the past. In fact, in some cases it might not be possible to recover at all:

- If the room was invite-only, your users will need to be re-invited.
- If the room no longer has any members at all, it'll be impossible to rejoin.
- The first user to rejoin will have to do so via an alias on a different server (or receive an invite from a user on a different server).

With all that being said, if you still want to try and recover the room:

1. If the room was `block`ed, you must unblock it on your server. This can be accomplished as follows:
 1. For safety reasons, shut down Synapse.
 2. In the database, run `DELETE FROM blocked_rooms WHERE room_id = ' !example@example.org' ;`
 - For caution: it's recommended to run this in a transaction: `BEGIN; DELETE ... ;`, verify you got 1 result, then `COMMIT;`.
 - The room ID is the same one supplied to the delete room API, not the Content Violation room.
 3. Restart Synapse.

This step is unnecessary if `block` was not set.

2. Any room aliases on your server that pointed to the deleted room may have been deleted, or redirected to the Content Violation room. These will need to be restored manually.
3. Users on your server that were in the deleted room will have been kicked from the room. Consider whether you want to update their membership (possibly via the [Edit Room Membership API](#)) or let them handle rejoining themselves.
4. If `new_room_user_id` was given, a 'Content Violation' will have been created. Consider whether you want to delete that room.

Make Room Admin API

Grants another user the highest power available to a local user who is in the room. If the user is not in the room, and it is not publicly joinable, then invite the user.

By default the server admin (the caller) is granted power, but another user can optionally be specified, e.g.:

```
POST /_synapse/admin/v1/rooms/<room_id_or_alias>/make_room_admin
{
  "user_id": "@foo@example.com"
}
```

Forward Extremities Admin API

Enables querying and deleting forward extremities from rooms. When a lot of forward extremities accumulate in a room, performance can become degraded. For details, see [#1760](#).

Check for forward extremities

To check the status of forward extremities for a room:

```
GET /_synapse/admin/v1/rooms/<room_id_or_alias>/forward_extremities
```

A response as follows will be returned:

```
{
  "count": 1,
  "results": [
    {
      "event_id": "$M5SP266vsnxctfwFgFLNceaCo3ujhRtg_NiiHabcdefg",
      "state_group": 439,
      "depth": 123,
      "received_ts": 1611263016761
    }
  ]
}
```

Deleting forward extremities

WARNING: Please ensure you know what you're doing and have read the related issue [#1760](#). Under no situations should this API be executed as an automated maintenance task!

If a room has lots of forward extremities, the extra can be deleted as follows:

```
DELETE /_synapse/admin/v1/rooms/<room_id_or_alias>/forward_extremities
```

A response as follows will be returned, indicating the amount of forward extremities that were deleted.

```
{  
  "deleted": 1  
}
```

Event Context API

This API lets a client find the context of an event. This is designed primarily to investigate abuse reports.

```
GET /_synapse/admin/v1/rooms/<room_id>/context/<event_id>
```

This API mimicks [GET /_matrix/client/r0/rooms/{roomId}/context/{eventId}](#). Please refer to the link for all details on parameters and response.

Example response:

```
{  
  "end": "t29-57_2_0_2",  
  "events_after": [  
    {  
      "content": {  
        "body": "This is an example text message",  
        "msgtype": "m.text",  
        "format": "org.matrix.custom.html",  
        "formatted_body": "<b>This is an example text message</b>"  
      },  
      "type": "m.room.message",  
      "event_id": "$143273582443PhrSn@example.org",  
      "room_id": "!636q39766251@example.com",  
      "sender": "@example@example.org",  
      "origin_server_ts": 1432735824653,  
      "unsigned": {  
        "age": 1234  
      }  
    }  
  ],  
  "event": {  
    "content": {  
      "body": "filename.jpg",  
      "info": {  
        "h": 398,  
        "w": 394,  
        "mimetype": "image/jpeg",  
        "size": 31037  
      },  
      "url": "mxc://example.org/JWEIFJgwEIhweiWJE",  
      "msgtype": "m.image"  
    },  
    "type": "m.room.message",  
    "event_id": "$f3h4d129462ha@example.com",  
    "room_id": "!636q39766251@example.com",  
    "sender": "@example@example.org",  
    "origin_server_ts": 1432735824653,  
    "unsigned": {  
      "age": 1234  
    }  
  },  
  "events_before": [  
    {  
      "content": {  
        "body": "something-important.doc",  
        "filename": "something-important.doc",  
        "info": {  
          "mimetype": "application/msword",  
          "size": 46144  
        },  
        "msgtype": "m.file",  
        "url": "mxc://example.org/FHyPlCeYUSFFxlgbQYZmoEoe"  
      },  
      "type": "m.room.message",  
      "event_id": "$143273582443PhrSn@example.org",  
      "room_id": "!636q39766251@example.com",  
      "sender": "@example@example.org",  
    }  
  ]  
}
```

```
"origin_server_ts": 1432735824653,
"unsigned": {
    "age": 1234
}
],
"start": "t27-54_2_0_2",
"state": [
{
    "content": {
        "creator": "@example:example.org",
        "room_version": "1",
        "m.federate": true,
        "predecessor": {
            "event_id": "$something:example.org",
            "room_id": "!oldroom:example.org"
        }
    },
    "type": "m.room.create",
    "event_id": "$143273582443PhrSn:example.org",
    "room_id": "!636q39766251:example.com",
    "sender": "@example:example.org",
    "origin_server_ts": 1432735824653,
    "unsigned": {
        "age": 1234
    },
    "state_key": ""
},
{
    "content": {
        "membership": "join",
        "avatar_url": "mxc://example.org/SEsfnsuifSDFSEF",
        "displayname": "Alice Margatroid"
    },
    "type": "m.room.member",
    "event_id": "$143273582443PhrSn:example.org",
    "room_id": "!636q39766251:example.com",
    "sender": "@example:example.org",
    "origin_server_ts": 1432735824653,
    "unsigned": {
        "age": 1234
    },
    "state_key": "@alice:example.org"
}
]
}
```

Server Notices

The API to send notices is as follows:

```
POST /_synapse/admin/v1/send_server_notice
```

or:

```
PUT /_synapse/admin/v1/send_server_notice/{txnId}
```

You will need to authenticate with an access token for an admin user.

When using the `PUT` form, retransmissions with the same transaction ID will be ignored in the same way as with `PUT`

```
/_matrix/client/r0/rooms/{roomId}/send/{eventType}/{txnId}.
```

The request body should look something like the following:

```
{
  "user_id": "@target_user:server_name",
  "content": {
    "msgtype": "m.text",
    "body": "This is my message"
  }
}
```

You can optionally include the following additional parameters:

- `type`: the type of event. Defaults to `m.room.message`.
- `state_key`: Setting this will result in a state event being sent.

Once the notice has been sent, the API will return the following response:

```
{
  "event_id": "<event_id>"
}
```

Note that server notices must be enabled in `homeserver.yaml` before this API can be used. See [the server notices documentation](#) for more information.

Users' media usage statistics

Returns information about all local media usage of users. Gives the possibility to filter them by time and user.

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

The API is:

```
GET /_synapse/admin/v1/statistics/users/media
```

A response body like the following is returned:

```
{
  "users": [
    {
      "displayname": "foo_user_0",
      "media_count": 2,
      "media_length": 134,
      "user_id": "@foo_user_0:test"
    },
    {
      "displayname": "foo_user_1",
      "media_count": 2,
      "media_length": 134,
      "user_id": "@foo_user_1:test"
    }
  ],
  "next_token": 3,
  "total": 10
}
```

To paginate, check for `next_token` and if present, call the endpoint again with `from` set to the value of `next_token`. This will return a new page.

If the endpoint does not return a `next_token` then there are no more reports to paginate through.

Parameters

The following parameters should be set in the URL:

- `limit`: string representing a positive integer - Is optional but is used for pagination, denoting the maximum number of items to return in this call. Defaults to `100`.
- `from`: string representing a positive integer - Is optional but used for pagination, denoting the offset in the returned results. This should be treated as an opaque value and not explicitly set to anything other than the return value of `next_token` from a previous call. Defaults to `0`.

- **order_by** - string - The method in which to sort the returned list of users. Valid values are:
 - **user_id** - Users are ordered alphabetically by **user_id**. This is the default.
 - **displayname** - Users are ordered alphabetically by **displayname**.
 - **media_length** - Users are ordered by the total size of uploaded media in bytes. Smallest to largest.
 - **media_count** - Users are ordered by number of uploaded media. Smallest to largest.
- **from_ts** - string representing a positive integer - Considers only files created at this timestamp or later. Unix timestamp in ms.
- **until_ts** - string representing a positive integer - Considers only files created at this timestamp or earlier. Unix timestamp in ms.
- **search_term** - string - Filter users by their user ID localpart **or** displayname. The search term can be found in any part of the string. Defaults to no filtering.
- **dir** - string - Direction of order. Either **f** for forwards or **b** for backwards. Setting this value to **b** will reverse the above sort order. Defaults to **f**.

Response

The following fields are returned in the JSON response body:

- **users** - An array of objects, each containing information about the user and their local media. Objects contain the following fields:
 - **displayname** - string - Displayname of this user.
 - **media_count** - integer - Number of uploaded media by this user.
 - **media_length** - integer - Size of uploaded media in bytes by this user.
 - **user_id** - string - Fully-qualified user ID (ex. `@user:server.com`).
- **next_token** - integer - Opaque value used for pagination. See above.
- **total** - integer - Total number of users after filtering.

Get largest rooms by size in database

Returns the 10 largest rooms and an estimate of how much space in the database they are taking.

This does not include the size of any associated media associated with the room.

Returns an error on SQLite.

Note: This uses the planner statistics from PostgreSQL to do the estimates, which means that the returned information can vary widely from reality. However, it should be enough to get a rough idea of where database disk space is going.

The API is:

```
GET /_synapse/admin/v1/statistics/database/rooms
```

A response body like the following is returned:

```
{  
  "rooms": [  
    {  
      "room_id": "!0GEhHVWSdvArJzumhm:matrix.org",  
      "estimated_size": 47325417353  
    }  
  ],  
}
```

Response

The following fields are returned in the JSON response body:

- **rooms** - An array of objects, sorted by largest room first. Objects contain the following fields:
 - **room_id** - string - The room ID.
 - **estimated_size** - integer - Estimated disk space used in bytes by the room in the database.

Added in Synapse 1.83.0

User Admin API

To use it, you will need to authenticate by providing an `access_token` for a server admin: see [Admin API](#).

Query User Account

This API returns information about a specific user account.

The api is:

```
GET /_synapse/admin/v2/users/<user_id>
```

It returns a JSON body like the following:

```
{
  "name": "@user@example.com",
  "displayname": "User", // can be null if not set
  "threepids": [
    {
      "medium": "email",
      "address": "<user_mail_1>",
      "added_at": 1586458409743,
      "validated_at": 1586458409743
    },
    {
      "medium": "email",
      "address": "<user_mail_2>",
      "added_at": 1586458409743,
      "validated_at": 1586458409743
    }
  ],
  "avatar_url": "<avatar_url>", // can be null if not set
  "is_guest": 0,
  "admin": 0,
  "deactivated": 0,
  "erased": false,
  "shadow_banned": 0,
  "creation_ts": 1560432506,
  "last_seen_ts": 1732919539393,
  "appservice_id": null,
  "consent_server_notice_sent": null,
  "consent_version": null,
  "consent_ts": null,
  "external_ids": [
    {
      "auth_provider": "<provider1>",
      "external_id": "<user_id_provider_1>"
    },
    {
      "auth_provider": "<provider2>",
      "external_id": "<user_id_provider_2>"
    }
  ],
  "user_type": null,
  "locked": false,
  "suspended": false
}
```

URL parameters:

- **user_id**: fully-qualified user id: for example, `@user:server.com`.

Create or modify account

This API allows an administrator to create or modify a user account with a specific `user_id`.

This api is:

```
PUT /_synapse/admin/v2/users/<user_id>
```

with a body of:

```
{
  "password": "user_password",
  "logout_devices": false,
  "displayname": "Alice Marigold",
  "avatar_url": "mxc://example.com/abcde12345",
  "threepids": [
    {
      "medium": "email",
      "address": "alice@example.com"
    },
    {
      "medium": "email",
      "address": "alice@domain.org"
    }
  ],
  "external_ids": [
    {
      "auth_provider": "example",
      "external_id": "12345"
    },
    {
      "auth_provider": "example2",
      "external_id": "abc54321"
    }
  ],
  "admin": false,
  "deactivated": false,
  "user_type": null,
  "locked": false
}
```

Returns HTTP status code:

- **201** - When a new user object was created.
- **200** - When a user was modified.

URL parameters:

- **user_id** - A fully-qualified user id. For example, `@user:server.com`.

Body parameters:

- **password** - **string**, optional. If provided, the user's password is updated and all devices are logged out, unless `logout_devices` is set to `false`.
- **logout_devices** - **bool**, optional, defaults to `true`. If set to `false`, devices aren't logged out even when `password` is provided.

- **displayname** - **string**, optional. If set to an empty string (""), the user's display name will be removed.
- **avatar_url** - **string**, optional. Must be a [MXC URI](#). If set to an empty string (""), the user's avatar is removed.
- **threepids** - **array**, optional. If provided, the user's third-party IDs (email, msisdn) are entirely replaced with the given list. Each item in the array is an object with the following fields:
 - **medium** - **string**, required. The type of third-party ID, either `email` or `msisdn` (phone number).
 - **address** - **string**, required. The third-party ID itself, e.g. `alice@example.com` for `email` or `447470274584` (for a phone number with country code "44") and `19254857364` (for a phone number with country code "1") for `msisdn`. Note: If a threepid is removed from a user via this option, Synapse will also attempt to remove that threepid from any identity servers it is aware has a binding for it.
- **external_ids** - **array**, optional. Allow setting the identifier of the external identity provider for SSO (Single sign-on). More details are in the configuration manual under the sections [sso](#) and [oidc_providers](#).
 - **auth_provider** - **string**, required. The unique, internal ID of the external identity provider. The same as `idp_id` from the homeserver configuration. If using OIDC, this value should be prefixed with `oidc-`. Note that no error is raised if the provided value is not in the homeserver configuration.
 - **external_id** - **string**, required. An identifier for the user in the external identity provider. When the user logs in to the identity provider, this must be the unique ID that they map to.
- **admin** - **bool**, optional, defaults to `false`. Whether the user is a homeserver administrator, granting them access to the Admin API, among other things.
- **deactivated** - **bool**, optional. If unspecified, deactivation state will be left unchanged.

Note:

- For the password field there is no strict check of the necessity for its presence. It is possible to have active users without a password, e.g. when authenticating with OIDC is configured. You must check yourself whether a password is required when reactivating a user or not.
- It is not possible to set a password if the config option `password_config.localdb_enabled` is set `false`. Users' passwords are wiped upon account deactivation, hence the need to set a new one here.

Note: a user cannot be erased with this API. For more details on deactivating and erasing users see [Deactivate Account](#).

- `locked` - **bool**, optional. If unspecified, locked state will be left unchanged.
- `user_type` - **string** or null, optional. If not provided, the user type will be not be changed. If `null` is given, the user type will be cleared. Other allowed options are: `bot` and `support`.

List Accounts

List Accounts (V2)

This API returns all local user accounts. By default, the response is ordered by ascending user ID.

```
GET /_synapse/admin/v2/users?from=0&limit=10&guests=false
```

A response body like the following is returned:

```
{
  "users": [
    {
      "name": "<user_id1>",
      "is_guest": 0,
      "admin": 0,
      "user_type": null,
      "deactivated": 0,
      "erased": false,
      "shadow_banned": 0,
      "displayname": "<User One>",
      "avatar_url": null,
      "creation_ts": 1560432668000,
      "locked": false
    },
    {
      "name": "<user_id2>",
      "is_guest": 0,
      "admin": 1,
      "user_type": null,
      "deactivated": 0,
      "erased": false,
      "shadow_banned": 0,
      "displayname": "<User Two>",
      "avatar_url": "<avatar_url>",
      "creation_ts": 1561550621000,
      "locked": false
    }
  ],
  "next_token": "100",
  "total": 200
}
```

To paginate, check for `next_token` and if present, call the endpoint again with `from` set to the value of `next_token`. This will return a new page.

If the endpoint does not return a `next_token` then there are no more users to paginate through.

Parameters

The following parameters should be set in the URL:

- `user_id` - Is optional and filters to only return users with user IDs that contain this value. This parameter is ignored when using the `name` parameter.
- `name` - Is optional and filters to only return users with user ID localparts **or** displaynames that contain this value.
- `guests` - string representing a bool - Is optional and if `false` will **exclude** guest users. Defaults to `true` to include guest users. This parameter is not supported when MSC3861 is enabled. [See #15582](#)
- `admins` - Optional flag to filter admins. If `true`, only admins are queried. If `false`, admins are excluded from the query. When the flag is absent (the default), **both** admins and non-admins are included in the search results.
- `deactivated` - string representing a bool - Is optional and if `true` will **include** deactivated users. Defaults to `false` to exclude deactivated users.
- `limit` - string representing a positive integer - Is optional but is used for pagination, denoting the maximum number of items to return in this call. Defaults to `100`.
- `from` - string representing a positive integer - Is optional but used for pagination, denoting the offset in the returned results. This should be treated as an opaque value and not explicitly set to anything other than the return value of `next_token` from a previous call. Defaults to `0`.
- `order_by` - The method by which to sort the returned list of users. If the ordered field has duplicates, the second order is always by ascending `name`, which guarantees a stable ordering. Valid values are:
 - `name` - Users are ordered alphabetically by `name`. This is the default.
 - `is_guest` - Users are ordered by `is_guest` status.
 - `admin` - Users are ordered by `admin` status.
 - `user_type` - Users are ordered alphabetically by `user_type`.
 - `deactivated` - Users are ordered by `deactivated` status.
 - `shadow_banned` - Users are ordered by `shadow_banned` status.
 - `displayname` - Users are ordered alphabetically by `displayname`.
 - `avatar_url` - Users are ordered alphabetically by avatar URL.

- `creation_ts` - Users are ordered by when the users was created in ms.
- `last_seen_ts` - Users are ordered by when the user was lastly seen in ms.
- `dir` - Direction of media order. Either `f` for forwards or `b` for backwards. Setting this value to `b` will reverse the above sort order. Defaults to `f`.
- `not_user_type` - Exclude certain user types, such as bot users, from the request. Can be provided multiple times. Possible values are `bot`, `support` or "empty string". "empty string" here means to exclude users without a type.
- `locked` - string representing a bool - Is optional and if `true` will **include** locked users. Defaults to `false` to exclude locked users. Note: Introduced in v1.93.

Caution. The database only has indexes on the columns `name` and `creation_ts`. This means that if a different sort order is used (`is_guest`, `admin`, `user_type`, `deactivated`, `shadow_banned`, `avatar_url` or `displayname`), this can cause a large load on the database, especially for large environments.

Response

The following fields are returned in the JSON response body:

- `users` - An array of objects, each containing information about an user. User objects contain the following fields:
 - `name` - string - Fully-qualified user ID (ex. `@user:server.com`).
 - `is_guest` - bool - Status if that user is a guest account.
 - `admin` - bool - Status if that user is a server administrator.
 - `user_type` - string - Type of the user. Normal users are type `None`. This allows user type specific behaviour. There are also types `support` and `bot`.
 - `deactivated` - bool - Status if that user has been marked as deactivated.
 - `erased` - bool - Status if that user has been marked as erased.
 - `shadow_banned` - bool - Status if that user has been marked as shadow banned.
 - `displayname` - string - The user's display name if they have set one.
 - `avatar_url` - string - The user's avatar URL if they have set one.
 - `creation_ts` - integer - The user's creation timestamp in ms.
 - `last_seen_ts` - integer - The user's last activity timestamp in ms.
 - `locked` - bool - Status if that user has been marked as locked. Note: Introduced in v1.93.
- `next_token`: string representing a positive integer - Indication for pagination. See above.
- `total` - integer - Total number of media.

Added in Synapse 1.93: the `locked` query parameter and response field.

List Accounts (V3)

This API returns all local user accounts (see v2). In contrast to v2, the query parameter `deactivated` is handled differently.

```
GET /_synapse/admin/v3/users
```

Parameters

- `deactivated` - Optional flag to filter deactivated users. If `true`, only deactivated users are returned. If `false`, deactivated users are excluded from the query. When the flag is absent (the default), users are not filtered by deactivation status.

Query current sessions for a user

This API returns information about the active sessions for a specific user.

The endpoints are:

```
GET /_synapse/admin/v1/whois/<user_id>
```

and:

```
GET /_matrix/client/r0/admin/whois/<userId>
```

See also: [Client Server API Whois](#).

It returns a JSON body like the following:

```
{
  "user_id": "<user_id>",
  "devices": {
    "": {
      "sessions": [
        {
          "connections": [
            {
              "ip": "1.2.3.4",
              "last_seen": 1417222374433,
              "user_agent": "Mozilla/5.0 ..."
            },
            {
              "ip": "1.2.3.10",
              "last_seen": 1417222374500,
              "user_agent": "Dalvik/2.1.0 ..."
            }
          ]
        }
      ]
    }
  }
}
```

`last_seen` is measured in milliseconds since the Unix epoch.

Deactivate Account

This API deactivates an account. It removes active access tokens, resets the password, and deletes third-party IDs (to prevent the user requesting a password reset).

It can also mark the user as GDPR-erased. This means messages sent by the user will still be visible by anyone that was in the room when these messages were sent, but hidden from users joining the room afterwards.

The api is:

```
POST /_synapse/admin/v1/deactivate/<user_id>
```

with a body of:

```
{
  "erase": true
}
```

The `erase` parameter is optional and defaults to `false`. An empty body may be passed for backwards compatibility.

The following actions are performed when deactivating an user:

- Try to unbind 3PIDs from the identity server
- Remove all 3PIDs from the homeserver
- Delete all devices and E2EE keys
- Delete all access tokens
- Delete all pushers
- Delete the password hash
- Removal from all rooms the user is a member of
- Remove the user from the user directory
- Reject all pending invites
- Remove all account validity information related to the user
- Remove the arbitrary data store known as *account data*. For example, this includes:
 - list of ignored users;
 - push rules;
 - secret storage keys; and
 - cross-signing keys.

The following additional actions are performed during deactivation if `erase` is set to `true`:

- Remove the user's display name
- Remove the user's avatar URL
- Mark the user as erased

The following actions are **NOT** performed. The list may be incomplete.

- Remove mappings of SSO IDs
- **Delete media uploaded** by user (included avatar images)
- Delete sent and received messages
- Remove the user's creation (registration) timestamp
- **Remove rate limit overrides**
- Remove from monthly active users
- Remove user's consent information (consent version and timestamp)

Reset password

Note: This API is disabled when MSC3861 is enabled. See [#15582](#)

Changes the password of another user. This will automatically log the user out of all their devices.

The api is:

```
POST /_synapse/admin/v1/reset_password/<user_id>
```

with a body of:

```
{
  "new_password": "<secret>",
  "logout_devices": true
}
```

The parameter `new_password` is required. The parameter `logout_devices` is optional and defaults to `true`.

Get whether a user is a server administrator or not

Note: This API is disabled when MSC3861 is enabled. See #15582

The api is:

```
GET /_synapse/admin/v1/users/<user_id>/admin
```

A response body like the following is returned:

```
{
  "admin": true
}
```

Change whether a user is a server administrator or not

Note: This API is disabled when MSC3861 is enabled. See #15582

Note that you cannot demote yourself.

The api is:

```
PUT /_synapse/admin/v1/users/<user_id>/admin
```

with a body of:

```
{
  "admin": true
}
```

List joined rooms of a user

Gets a list of all `room_id` that a specific `user_id` is joined to and is a member of (participating in).

The API is:

```
GET /_synapse/admin/v1/users/<user_id>/joined_rooms
```

A response body like the following is returned:

```
{
  "joined_rooms": [
    "!DuGcnbhHGaSZQoNQR:matrix.org",
    "!ZtSaPCawyWtxfWiIy:matrix.org"
  ],
  "total": 2
}
```

The server returns the list of rooms of which the user and the server are member. If the user is local, all the rooms of which the user is member are returned.

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.

Response

The following fields are returned in the JSON response body:

- `joined_rooms` - An array of `room_id`.
- `total` - Number of rooms.

Get the number of invites sent by the user

Fetches the number of invites sent by the provided user ID across all rooms after the given timestamp.

```
GET /_synapse/admin/v1/users/$user_id/sent_invite_count
```

Parameters

The following parameters should be set in the URL:

- `user_id`: fully qualified: for example, `@user:server.com`

The following should be set as query parameters in the URL:

- `from_ts`: int, required. A timestamp in ms from the unix epoch. Only invites sent at or after the provided timestamp will be returned. This works by comparing the provided timestamp to the `received_ts` column in the `events` table. Note:

<https://currentmillis.com/> is a useful tool for converting dates into timestamps and vice versa.

A response body like the following is returned:

```
{
  "invite_count": 30
}
```

Added in Synapse 1.122.0

Get the cumulative number of rooms a user has joined after a given timestamp

Fetches the number of rooms that the user joined after the given timestamp, even if they have subsequently left/been banned from those rooms.

```
GET /_synapse/admin/v1/users/$<user_id>/cumulative_joined_room_count
```

Parameters

The following parameters should be set in the URL:

- `user_id`: fully qualified: for example, `@user:server.com`

The following should be set as query parameters in the URL:

- `from_ts`: int, required. A timestamp in ms from the unix epoch. Only invites sent at or after the provided timestamp will be returned. This works by comparing the provided timestamp to the `received_ts` column in the `events` table. Note:

<https://currentmillis.com/> is a useful tool for converting dates into timestamps and vice versa.

A response body like the following is returned:

```
{
  "cumulative_joined_room_count": 30
}
```

Added in Synapse 1.122.0

Account Data

Gets information about account data for a specific `user_id`.

The API is:

```
GET /_synapse/admin/v1/users/<user_id>/accountdata
```

A response body like the following is returned:

```
{
  "account_data": {
    "global": {
      "m.secret_storage.key.LmIGHTg5W": {
        "algorithm": "m.secret_storage.v1.aes-hmac-sha2",
        "iv": "fwjNZatxg==",
        "mac": "eWh9kNnLWZUN0gnc="
      },
      "im.vector.hide_profile": {
        "hide_profile": true
      },
      "org.matrix.preview_urls": {
        "disable": false
      },
      "im.vector.riot.breadcrumb_rooms": {
        "rooms": [
          "!LxcBDAsDUVAfJDEo:matrix.org",
          "!MAhRxqasbItj0qxu:matrix.org"
        ]
      },
      "m.accepted_terms": {
        "accepted": [
          "https://example.org/somewhere/privacy-1.2-en.html",
          "https://example.org/somewhere/terms-2.0-en.html"
        ]
      },
      "im.vector.setting.breadcrumbs": {
        "recent_rooms": [
          "!MAhRxqasbItqxuEt:matrix.org",
          "!ZtSaPCawyWtxiImy:matrix.org"
        ]
      }
    },
    "rooms": {
      "!GUdfZSHUJibpiVqHYd:matrix.org": {
        "m.fully_read": {
          "event_id": "$156334540fYIhZ:matrix.org"
        }
      },
      "!t0Zw00iqwCYQkLhV:matrix.org": {
        "m.fully_read": {
          "event_id": "$xjsIyp4_NaVl2yPvIZs_k1Jl8tsC_Sp23wjqXPno"
        }
      }
    }
  }
}
```

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.

Response

The following fields are returned in the JSON response body:

- `account_data` - A map containing the account data for the user
 - `global` - A map containing the global account data for the user
 - `rooms` - A map containing the account data per room for the user

User media

List media uploaded by a user

Gets a list of all local media that a specific `user_id` has created. These are media that the user has uploaded themselves ([local media](#)), as well as [URL preview images](#) requested by the user if the [feature is enabled](#).

By default, the response is ordered by descending creation date and ascending media ID. The newest media is on top. You can change the order with parameters `order_by` and `dir`.

The API is:

```
GET /_synapse/admin/v1/users/<user_id>/media
```

A response body like the following is returned:

```
{
  "media": [
    {
      "created_ts": 100400,
      "last_access_ts": null,
      "media_id": "qXhyRzulkwLsNHTbpHreuEgo",
      "media_length": 67,
      "media_type": "image/png",
      "quarantined_by": null,
      "safe_from_quarantine": false,
      "upload_name": "test1.png"
    },
    {
      "created_ts": 200400,
      "last_access_ts": null,
      "media_id": "FHfiSnzoINDatrXHQIXBtahw",
      "media_length": 67,
      "media_type": "image/png",
      "quarantined_by": null,
      "safe_from_quarantine": false,
      "upload_name": "test2.png"
    },
    {
      "created_ts": 300400,
      "last_access_ts": 300700,
      "media_id": "BzYNLRUgGHphBkdKGbzXwbjX",
      "media_length": 1337,
      "media_type": "application/octet-stream",
      "quarantined_by": null,
      "safe_from_quarantine": false,
      "upload_name": null
    }
  ],
  "next_token": 3,
  "total": 2
}
```

To paginate, check for `next_token` and if present, call the endpoint again with `from` set to the value of `next_token`. This will return a new page.

If the endpoint does not return a `next_token` then there are no more reports to paginate through.

Parameters

The following parameters should be set in the URL:

- `user_id` - string - fully qualified: for example, `@user:server.com`.
- `limit`: string representing a positive integer - Is optional but is used for pagination, denoting the maximum number of items to return in this call. Defaults to `100`.
- `from`: string representing a positive integer - Is optional but used for pagination, denoting the offset in the returned results. This should be treated as an opaque value

and not explicitly set to anything other than the return value of `next_token` from a previous call. Defaults to `0`.

- `order_by` - The method by which to sort the returned list of media. If the ordered field has duplicates, the second order is always by ascending `media_id`, which guarantees a stable ordering. Valid values are:
 - `media_id` - Media are ordered alphabetically by `media_id`.
 - `upload_name` - Media are ordered alphabetically by name the media was uploaded with.
 - `created_ts` - Media are ordered by when the content was uploaded in ms. Smallest to largest. This is the default.
 - `last_access_ts` - Media are ordered by when the content was last accessed in ms. Smallest to largest.
 - `media_length` - Media are ordered by length of the media in bytes. Smallest to largest.
 - `media_type` - Media are ordered alphabetically by MIME-type.
 - `quarantined_by` - Media are ordered alphabetically by the user ID that initiated the quarantine request for this media.
 - `safe_from_quarantine` - Media are ordered by the status if this media is safe from quarantining.
- `dir` - Direction of media order. Either `f` for forwards or `b` for backwards. Setting this value to `b` will reverse the above sort order. Defaults to `f`.

If neither `order_by` nor `dir` is set, the default order is newest media on top (corresponds to `order_by = created_ts` and `dir = b`).

Caution. The database only has indexes on the columns `media_id`, `user_id` and `created_ts`. This means that if a different sort order is used (`upload_name`, `last_access_ts`, `media_length`, `media_type`, `quarantined_by` or `safe_from_quarantine`), this can cause a large load on the database, especially for large environments.

Response

The following fields are returned in the JSON response body:

- `media` - An array of objects, each containing information about a media. Media objects contain the following fields:
 - `created_ts` - integer - Timestamp when the content was uploaded in ms.
 - `last_access_ts` - integer or null - Timestamp when the content was last accessed in ms. Null if there was no access, yet.
 - `media_id` - string - The id used to refer to the media. Details about the format are documented under [media repository](#).
 - `media_length` - integer - Length of the media in bytes.
 - `media_type` - string - The MIME-type of the media.

- `quarantined_by` - string or null - The user ID that initiated the quarantine request for this media. Null if not quarantined.
- `safe_from_quarantine` - bool - Status if this media is safe from quarantining.
- `upload_name` - string or null - The name the media was uploaded with. Null if not provided during upload.
- `next_token`: integer - Indication for pagination. See above.
- `total` - integer - Total number of media.

Delete media uploaded by a user

This API deletes the *local* media from the disk of your own server that a specific `user_id` has created. This includes any local thumbnails.

This API will not affect media that has been uploaded to external media repositories (e.g <https://github.com/turt2live/matrix-media-repo/>).

By default, the API deletes media ordered by descending creation date and ascending media ID. The newest media is deleted first. You can change the order with parameters `order_by` and `dir`. If no `limit` is set the API deletes `100` files per request.

The API is:

```
DELETE /_synapse/admin/v1/users/<user_id>/media
```

A response body like the following is returned:

```
{
  "deleted_media": [
    "abcdefghijklmnopqrstuvwxyz"
  ],
  "total": 1
}
```

The following fields are returned in the JSON response body:

- `deleted_media`: an array of strings - List of deleted `media_id`
- `total`: integer - Total number of deleted `media_id`

Note: There is no `next_token`. This is not useful for deleting media, because after deleting media the remaining media have a new order.

Parameters

This API has the same parameters as [List media uploaded by a user](#). With the parameters you can for example limit the number of files to delete at once or delete largest/smallest or newest/oldest files first.

Login as a user

Note: This API is disabled when MSC3861 is enabled. See #15582

Get an access token that can be used to authenticate as that user. Useful for when admins wish to do actions on behalf of a user.

The API is:

```
POST /_synapse/admin/v1/users/<user_id>/login
{}
```

An optional `valid_until_ms` field can be specified in the request body as an integer timestamp that specifies when the token should expire. By default tokens do not expire. Note that this API does not allow a user to login as themselves (to create more tokens).

A response body like the following is returned:

```
{
  "access_token": "<opaque_access_token_string>"
}
```

This API does *not* generate a new device for the user, and so will not appear in their `/devices` list, and in general the target user should not be able to tell they have been logged in as.

To expire the token call the standard `/logout` API with the token.

Note: The token will expire if the `admin` user calls `/logout/all` from any of their devices, but the token will *not* expire if the target user does the same.

Allow replacing master cross-signing key without User-Interactive Auth

This endpoint is not intended for server administrator usage; we describe it here for completeness.

This API temporarily permits a user to replace their master cross-signing key without going through [user-interactive authentication](#) (UIA). This is useful when Synapse has delegated its authentication to the [Matrix Authentication Service](#); as Synapse cannot perform UIA is not possible in these circumstances.

The API is

```
POST
/_synapse/admin/v1/users/<user_id>/_allow_cross_signing_replacement_without_uia
{}
```

If the user does not exist, or does exist but has no master cross-signing key, this will return with status code `404 Not Found`.

Otherwise, a response body like the following is returned, with status `200 OK`:

```
{  
  "updatable_without_uia_before_ms": 1234567890  
}
```

The response body is a JSON object with a single field:

- `updatable_without_uia_before_ms`: integer. The timestamp in milliseconds before which the user is permitted to replace their cross-signing key without going through UIA.

Added in Synapse 1.97.0.

User devices

List all devices

Gets information about all devices for a specific `user_id`.

The API is:

```
GET /_synapse/admin/v2/users/<user_id>/devices
```

A response body like the following is returned:

```
{
  "devices": [
    {
      "device_id": "QBUAZIFURK",
      "display_name": "android",
      "last_seen_ip": "1.2.3.4",
      "last_seen_user_agent": "Mozilla/5.0 (X11; Linux x86_64; rv:103.0) Gecko/20100101 Firefox/103.0",
      "last_seen_ts": 1474491775024,
      "user_id": "<user_id>"
    },
    {
      "device_id": "AUIECTSrnd",
      "display_name": "ios",
      "last_seen_ip": "1.2.3.5",
      "last_seen_user_agent": "Mozilla/5.0 (X11; Linux x86_64; rv:103.0) Gecko/20100101 Firefox/103.0",
      "last_seen_ts": 1474491775025,
      "user_id": "<user_id>"
    }
  ],
  "total": 2
}
```

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.

Response

The following fields are returned in the JSON response body:

- `devices` - An array of objects, each containing information about a device. Device objects contain the following fields:
 - `device_id` - Identifier of device.
 - `display_name` - Display name set by the user for this device. Absent if no name has been set.
 - `last_seen_ip` - The IP address where this device was last seen. (May be a few minutes out of date, for efficiency reasons).
 - `last_seen_user_agent` - The user agent of the device when it was last seen. (May be a few minutes out of date, for efficiency reasons).
 - `last_seen_ts` - The timestamp (in milliseconds since the unix epoch) when this device was last seen. (May be a few minutes out of date, for efficiency reasons).
 - `user_id` - Owner of device.
- `total` - Total number of user's devices.

Create a device

Creates a new device for a specific `user_id` and `device_id`. Does nothing if the `device_id` exists already.

The API is:

```
POST /_synapse/admin/v2/users/<user_id>/devices
{
  "device_id": "QBUAZIFURK"
}
```

An empty JSON dict is returned.

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.

The following fields are required in the JSON request body:

- `device_id` - The device ID to create.

Delete multiple devices

Deletes the given devices for a specific `user_id`, and invalidates any access token associated with them.

The API is:

```
POST /_synapse/admin/v2/users/<user_id>/delete_devices
{
  "devices": [
    "QBUAZIFURK",
    "AUIECTSrnd"
  ]
}
```

An empty JSON dict is returned.

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.

The following fields are required in the JSON request body:

- **devices** - The list of device IDs to delete.

Show a device

Gets information on a single device, by `device_id` for a specific `user_id`.

The API is:

```
GET /_synapse/admin/v2/users/<user_id>/devices/<device_id>
```

A response body like the following is returned:

```
{
  "device_id": "<device_id>",
  "display_name": "android",
  "last_seen_ip": "1.2.3.4",
  "last_seen_user_agent": "Mozilla/5.0 (X11; Linux x86_64; rv:103.0) Gecko/20100101 Firefox/103.0",
  "last_seen_ts": 1474491775024,
  "user_id": "<user_id>"
}
```

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.
- `device_id` - The device to retrieve.

Response

The following fields are returned in the JSON response body:

- `device_id` - Identifier of device.
- `display_name` - Display name set by the user for this device. Absent if no name has been set.
- `last_seen_ip` - The IP address where this device was last seen. (May be a few minutes out of date, for efficiency reasons).
 - `last_seen_user_agent` - The user agent of the device when it was last seen. (May be a few minutes out of date, for efficiency reasons).
- `last_seen_ts` - The timestamp (in milliseconds since the unix epoch) when this device was last seen. (May be a few minutes out of date, for efficiency reasons).
- `user_id` - Owner of device.

Update a device

Updates the metadata on the given `device_id` for a specific `user_id`.

The API is:

```
PUT /_synapse/admin/v2/users/<user_id>/devices/<device_id>
{
  "display_name": "My other phone"
}
```

An empty JSON dict is returned.

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.
- `device_id` - The device to update.

The following fields are required in the JSON request body:

- `display_name` - The new display name for this device. If not given, the display name is unchanged.

Delete a device

Deletes the given `device_id` for a specific `user_id`, and invalidates any access token associated with it.

The API is:

```
DELETE /_synapse/admin/v2/users/<user_id>/devices/<device_id>
{}
```

An empty JSON dict is returned.

Parameters

The following parameters should be set in the URL:

- `user_id` - fully qualified: for example, `@user:server.com`.
- `device_id` - The device to delete.

List all pushers

Gets information about all pushers for a specific `user_id`.

The API is:

```
GET /_synapse/admin/v1/users/<user_id>/pushers
```

A response body like the following is returned:

```
{  
  "pushers": [  
    {  
      "app_display_name": "HTTP Push Notifications",  
      "app_id": "m.http",  
      "data": {  
        "url": "example.com"  
      },  
      "device_display_name": "pushy push",  
      "kind": "http",  
      "lang": "None",  
      "profile_tag": "",  
      "pushkey": "a@example.com"  
    }  
  ],  
  "total": 1  
}
```

Parameters

The following parameters should be set in the URL:

- **user_id** - fully qualified: for example, `@user:server.com`.

Response

The following fields are returned in the JSON response body:

- **pushers** - An array containing the current pushers for the user
 - **app_display_name** - string - A string that will allow the user to identify what application owns this pusher.
 - **app_id** - string - This is a reverse-DNS style identifier for the application. Max length, 64 chars.
 - **data** - A dictionary of information for the pusher implementation itself.
 - **url** - string - Required if **kind** is `http`. The URL to use to send notifications to.
 - **format** - string - The format to use when sending notifications to the Push Gateway.
 - **device_display_name** - string - A string that will allow the user to identify what device owns this pusher.

- **profile_tag** - string - This string determines which set of device specific rules this pusher executes.
- **kind** - string - The kind of pusher. "http" is a pusher that sends HTTP pokes.
- **lang** - string - The preferred language for receiving notifications (e.g. 'en' or 'en-US')
- **profile_tag** - string - This string determines which set of device specific rules this pusher executes.
- **pushkey** - string - This is a unique identifier for this pusher. Max length, 512 bytes.
- **total** - integer - Number of pushers.

See also the [Client-Server API Spec on pushers](#).

Controlling whether a user is shadow-banned

Shadow-banning is a useful tool for moderating malicious or egregiously abusive users. A shadow-banned user receives successful responses to their client-server API requests, but the events are not propagated into rooms. This can be an effective tool as it (hopefully) takes longer for the user to realise they are being moderated before pivoting to another account.

Shadow-banning a user should be used as a tool of last resort and may lead to confusing or broken behaviour for the client. A shadow-banned user will not receive any notification and it is generally more appropriate to ban or kick abusive users. A shadow-banned user will be unable to contact anyone on the server.

To shadow-ban a user the API is:

```
POST /_synapse/admin/v1/users/<user_id>/shadow_ban
```

To un-shadow-ban a user the API is:

```
DELETE /_synapse/admin/v1/users/<user_id>/shadow_ban
```

An empty JSON dict is returned in both cases.

Parameters

The following parameters should be set in the URL:

- `user_id` - The fully qualified MXID: for example, `@user:server.com`. The user must be local.

Override ratelimiting for users

This API allows to override or disable ratelimiting for a specific user. There are specific APIs to set, get and delete a ratelimit.

Get status of ratelimit

The API is:

```
GET /_synapse/admin/v1/users/<user_id>/override_ratelimit
```

A response body like the following is returned:

```
{  
  "messages_per_second": 0,  
  "burst_count": 0  
}
```

Parameters

The following parameters should be set in the URL:

- `user_id` - The fully qualified MXID: for example, `@user:server.com`. The user must be local.

Response

The following fields are returned in the JSON response body:

- `messages_per_second` - integer - The number of actions that can be performed in a second. `0` mean that ratelimiting is disabled for this user.
- `burst_count` - integer - How many actions that can be performed before being limited.

If **no** custom ratelimit is set, an empty JSON dict is returned.

```
{}
```

Set ratelimit

The API is:

```
POST /_synapse/admin/v1/users/<user_id>/override_ratelimit
```

A response body like the following is returned:

```
{  
  "messages_per_second": 0,  
  "burst_count": 0  
}
```

Parameters

The following parameters should be set in the URL:

- `user_id` - The fully qualified MXID: for example, `@user:server.com`. The user must be local.

Body parameters:

- `messages_per_second` - positive integer, optional. The number of actions that can be performed in a second. Defaults to `0`.
- `burst_count` - positive integer, optional. How many actions that can be performed before being limited. Defaults to `0`.

To disable users' ratelimit set both values to `0`.

Response

The following fields are returned in the JSON response body:

- `messages_per_second` - integer - The number of actions that can be performed in a second.
- `burst_count` - integer - How many actions that can be performed before being limited.

Delete ratelimit

The API is:

```
DELETE /_synapse/admin/v1/users/<user_id>/override_ratelimit
```

An empty JSON dict is returned.

```
[]
```

Parameters

The following parameters should be set in the URL:

- `user_id` - The fully qualified MXID: for example, `@user:server.com`. The user must be local.

Check username availability

Checks to see if a username is available, and valid, for the server. See [the client-server API](#) for more information.

This endpoint will work even if registration is disabled on the server, unlike `/_matrix/client/r0/register/available`.

The API is:

```
GET /_synapse/admin/v1/username_available?username=$localpart
```

The request and response format is the same as the [/_matrix/client/r0/register/available](#) API.

Find a user based on their ID in an auth provider

The API is:

```
GET /_synapse/admin/v1/auth_providers/$provider/users/$external_id
```

When a user matched the given ID for the given provider, an HTTP code `200` with a response body like the following is returned:

```
{
  "user_id": "@hello@example.org"
}
```

Parameters

The following parameters should be set in the URL:

- `provider` - The ID of the authentication provider, as advertised by the [GET `/_matrix/client/v3/login`](#) API in the `m.login.sso` authentication method.
- `external_id` - The user ID from the authentication provider. Usually corresponds to the `sub` claim for OIDC providers, or to the `uid` attestation for SAML2 providers.

The `external_id` may have characters that are not URL-safe (typically `/`, `:` or `@`), so it is advised to URL-encode those parameters.

Errors

Returns a **404** HTTP status code if no user was found, with a response body like this:

```
{  
  "errcode": "M_NOT_FOUND",  
  "error": "User not found"  
}
```

Added in Synapse 1.68.0.

Find a user based on their Third Party ID (ThreePID or 3PID)

The API is:

```
GET /_synapse/admin/v1/threepid/$medium/users/$address
```

When a user matched the given address for the given medium, an HTTP code **200** with a response body like the following is returned:

```
{  
  "user_id": "@hello@example.org"  
}
```

Parameters

The following parameters should be set in the URL:

- **medium** - Kind of third-party ID, either `email` or `msisdn`.
- **address** - Value of the third-party ID.

The `address` may have characters that are not URL-safe, so it is advised to URL-encode those parameters.

Errors

Returns a **404** HTTP status code if no user was found, with a response body like this:

```
{  
  "errcode": "M_NOT_FOUND",  
  "error": "User not found"  
}
```

Added in Synapse 1.72.0.

Redact all the events of a user

This endpoint allows an admin to redact the events of a given user. There are no restrictions on redactions for a local user. By default, we puppet the user who sent the message to redact it themselves. Redactions for non-local users are issued using the admin user, and will fail in rooms where the admin user is not admin/does not have the specified power level to issue redactions.

The API is

```
POST /_synapse/admin/v1/user/$user_id/redact
{
  "rooms": ["!roomid1", "!roomid2"]
}
```

If an empty list is provided as the key for `rooms`, all events in all the rooms the user is member of will be redacted, otherwise all the events in the rooms provided in the request will be redacted.

The API starts redaction process running, and returns immediately with a JSON body with a redact id which can be used to query the status of the redaction process:

```
{
  "redact_id": "<opaque id>"
}
```

Parameters

The following parameters should be set in the URL:

- `user_id` - The fully qualified MXID of the user: for example, `@user:server.com`.

The following JSON body parameter must be provided:

- `rooms` - A list of rooms to redact the user's events in. If an empty list is provided all events in all rooms the user is a member of will be redacted

Added in Synapse 1.116.0.

The following JSON body parameters are optional:

- `reason` - Reason the redaction is being requested, ie "spam", "abuse", etc. This will be included in each redaction event, and be visible to users.
- `limit` - a limit on the number of the user's events to search for ones that can be redacted (events are redacted newest to oldest) in each room, defaults to 1000 if not provided

Check the status of a redaction process

It is possible to query the status of the background task for redacting a user's events. The status can be queried up to 24 hours after completion of the task, or until Synapse is restarted (whichever happens first).

The API is:

```
GET /_synapse/admin/v1/user/redact_status/$redact_id
```

A response body like the following is returned:

```
{  
  "status": "active",  
  "failed_redactions": []  
}
```

Parameters

The following parameters should be set in the URL:

- **redact_id** - string - The ID for this redaction process, provided when the redaction was requested.

Response

The following fields are returned in the JSON response body:

- **status** - string - one of scheduled/active/completed/failed, indicating the status of the redaction job
- **failed_redactions** - dictionary - the keys of the dict are event ids the process was unable to redact, if any, and the values are the corresponding error that caused the redaction to fail

Added in Synapse 1.116.0.

Version API

This API returns the running Synapse version. This is useful when a Synapse instance is behind a proxy that does not forward the 'Server' header (which also contains Synapse version information).

The api is:

```
GET /_synapse/admin/v1/server_version
```

It returns a JSON body like the following:

```
{  
    "server_version": "0.99.2rc1 (b=develop, abcdef123)"  
}
```

Changed in Synapse 1.94.0: The `python_version` key was removed from the response body.

Federation API

This API allows a server administrator to manage Synapse's federation with other homeservers.

Note: This API is new, experimental and "subject to change".

List of destinations

This API gets the current destination retry timing info for all remote servers.

The list contains all the servers with which the server federates, regardless of whether an error occurred or not. If an error occurs, it may take up to 20 minutes for the error to be displayed here, as a complete retry must have failed.

The API is:

A standard request with no filtering:

```
GET /_synapse/admin/v1/federation/destinations
```

A response body like the following is returned:

```
{
  "destinations": [
    {
      "destination": "matrix.org",
      "retry_last_ts": 1557332397936,
      "retry_interval": 3000000,
      "failure_ts": 1557329397936,
      "last_successful_stream_ordering": null
    }
  ],
  "total": 1
}
```

To paginate, check for `next_token` and if present, call the endpoint again with `from` set to the value of `next_token`. This will return a new page.

If the endpoint does not return a `next_token` then there are no more destinations to paginate through.

Parameters

The following query parameters are available:

- **from** - Offset in the returned list. Defaults to `0`.
- **limit** - Maximum amount of destinations to return. Defaults to `100`.
- **order_by** - The method in which to sort the returned list of destinations. Valid values are:
 - **destination** - Destinations are ordered alphabetically by remote server name. This is the default.
 - **retry_last_ts** - Destinations are ordered by time of last retry attempt in ms.
 - **retry_interval** - Destinations are ordered by how long until next retry in ms.
 - **failure_ts** - Destinations are ordered by when the server started failing in ms.
 - **last_successful_stream_ordering** - Destinations are ordered by the stream ordering of the most recent successfully-sent PDU.
- **dir** - Direction of room order. Either `f` for forwards or `b` for backwards. Setting this value to `b` will reverse the above sort order. Defaults to `f`.

Caution: The database only has an index on the column `destination`. This means that if a different sort order is used, this can cause a large load on the database, especially for large environments.

Response

The following fields are returned in the JSON response body:

- **destinations** - An array of objects, each containing information about a destination. Destination objects contain the following fields:
 - **destination** - string - Name of the remote server to federate.
 - **retry_last_ts** - integer - The last time Synapse tried and failed to reach the remote server, in ms. This is `0` if the last attempt to communicate with the remote server was successful.
 - **retry_interval** - integer - How long since the last time Synapse tried to reach the remote server before trying again, in ms. This is `0` if no further retrying occurring.
 - **failure_ts** - nullable integer - The first time Synapse tried and failed to reach the remote server, in ms. This is `null` if communication with the remote server has never failed.
 - **last_successful_stream_ordering** - nullable integer - The stream ordering of the most recent successfully-sent PDU to this destination, or `null` if this information has not been tracked yet.
- **next_token** - string representing a positive integer - Indication for pagination. See above.
- **total** - integer - Total number of destinations.

Destination Details API

This API gets the retry timing info for a specific remote server.

The API is:

```
GET /_synapse/admin/v1/federation/destinations/<destination>
```

A response body like the following is returned:

```
{
  "destination": "matrix.org",
  "retry_last_ts": 1557332397936,
  "retry_interval": 3000000,
  "failure_ts": 1557329397936,
  "last_successful_stream_ordering": null
}
```

Parameters

The following parameters should be set in the URL:

- **destination** - Name of the remote server.

Response

The response fields are the same like in the **destinations** array in [List of destinations](#) response.

Destination rooms

This API gets the rooms that federate with a specific remote server.

The API is:

```
GET /_synapse/admin/v1/federation/destinations/<destination>/rooms
```

A response body like the following is returned:

```
{
  "rooms": [
    {
      "room_id": "!0GEhHVWSdvArJzumhm:matrix.org",
      "stream_ordering": 8326
    },
    {
      "room_id": "!xYvNcQPhnkrdUmYczI:matrix.org",
      "stream_ordering": 93534
    }
  ],
  "total": 2
}
```

To paginate, check for `next_token` and if present, call the endpoint again with `from` set to the value of `next_token`. This will return a new page.

If the endpoint does not return a `next_token` then there are no more destinations to paginate through.

Parameters

The following parameters should be set in the URL:

- `destination` - Name of the remote server.

The following query parameters are available:

- `from` - Offset in the returned list. Defaults to `0`.
- `limit` - Maximum amount of destinations to return. Defaults to `100`.
- `dir` - Direction of room order by `room_id`. Either `f` for forwards or `b` for backwards. Defaults to `f`.

Response

The following fields are returned in the JSON response body:

- `rooms` - An array of objects, each containing information about a room. Room objects contain the following fields:
 - `room_id` - string - The ID of the room.
 - `stream_ordering` - integer - The stream ordering of the most recent successfully-sent **PDU** to this destination in this room.
- `next_token`: string representing a positive integer - Indication for pagination. See above.
- `total` - integer - Total number of destinations.

Reset connection timeout

Synapse makes federation requests to other homeservers. If a federation request fails, Synapse will mark the destination homeserver as offline, preventing any future requests to that server for a "cooldown" period. This period grows over time if the server continues to fail its responses ([exponential backoff](#)).

Admins can cancel the cooldown period with this API.

This API resets the retry timing for a specific remote server and tries to connect to the remote server again. It does not wait for the next `retry_interval`. The connection must have previously run into an error and `retry_last_ts` ([Destination Details API](#)) must not be equal to `0`.

The connection attempt is carried out in the background and can take a while even if the API already returns the http status 200.

The API is:

```
POST /_synapse/admin/v1/federation/destinations/<destination>/reset_connection
{}
```

Parameters

The following parameters should be set in the URL:

- `destination` - Name of the remote server.

Using the synapse manhole

The "manhole" allows server administrators to access a Python shell on a running Synapse installation. This is a very powerful mechanism for administration and debugging.

Security Warning

Note that this will give administrative access to synapse to **all users** with shell access to the server. It should therefore **not** be enabled in environments where untrusted users have shell access.

Configuring the manhole

To enable it, first add the `manhole` listener configuration in your `homeserver.yaml`. You can find information on how to do that in the [configuration manual](#). The configuration is slightly different if you're using docker.

Docker config

If you are using Docker, set `bind_addresses` to `['0.0.0.0']` as shown:

```
listeners:
- port: 9000
  bind_addresses: ['0.0.0.0']
  type: manhole
```

When using `docker run` to start the server, you will then need to change the command to the following to include the `manhole` port forwarding. The `-p 127.0.0.1:9000:9000` below is important: it ensures that access to the `manhole` is only possible for local users.

```
docker run -d --name synapse \
--mount type=volume,src=synapse-data,dst=/data \
-p 8008:8008 \
-p 127.0.0.1:9000:9000 \
vectorim/synapse:latest
```

Native config

If you are not using docker, set `bind_addresses` to `['::1', '127.0.0.1']` as shown. The `bind_addresses` in the example below is important: it ensures that access to the `manhole` is only possible for local users).

```
listeners:
  - port: 9000
    bind_addresses: ['::1', '127.0.0.1']
    type: manhole
```

Security settings

The following config options are available:

- `username` - The username for the manhole (defaults to `matrix`)
- `password` - The password for the manhole (defaults to `rabbithole`)
- `ssh_priv_key` - The path to a private SSH key (defaults to a hardcoded value)
- `ssh_pub_key` - The path to a public SSH key (defaults to a hardcoded value)

For example:

```
manhole_settings:
  username: manhole
  password: mypassword
  ssh_priv_key: "/home/synapse/manhole_keys/id_rsa"
  ssh_pub_key: "/home/synapse/manhole_keys/id_rsa.pub"
```

Accessing synapse manhole

Then restart synapse, and point an ssh client at port 9000 on localhost, using the username and password configured in `homeserver.yaml` - with the default configuration, this would be:

```
ssh -p9000 matrix@localhost
```

Then enter the password when prompted (the default is `rabbithole`).

This gives a Python REPL in which `hs` gives access to the `synapse.server.HomeServer` object - which in turn gives access to many other parts of the process.

Note that, prior to Synapse 1.41, any call which returns a coroutine will need to be wrapped in `ensureDeferred`.

As a simple example, retrieving an event from the database:

```
>>> from twisted.internet import defer
>>>
defer.ensureDeferred(hs.get_datastores().main.get_event('$1416420717069yeQaw:matrix')
<Deferred at 0x7ff253fc6998 current result: <FrozenEvent
event_id='\$1416420717069yeQaw:matrix.org', type='m.room.create', state_key=''>
```

How to monitor Synapse metrics using Prometheus

1. Install Prometheus:

Follow instructions at <http://prometheus.io/docs/introduction/install/>

2. Enable Synapse metrics:

In `homeserver.yaml`, make sure `enable_metrics` is set to `True`.

3. Enable the `/_synapse/metrics` Synapse endpoint that Prometheus uses to collect data:

There are two methods of enabling the metrics endpoint in Synapse.

The first serves the metrics as a part of the usual web server and can be enabled by adding the `metrics` resource to the existing listener as such as in this example:

```
listeners:
  - port: 8008
    tls: false
    type: http
    x_forwarded: true
    bind_addresses: ['::1', '127.0.0.1']

  resources:
    # added "metrics" in this line
    - names: [client, federation, metrics]
      compress: false
```

This provides a simple way of adding metrics to your Synapse installation, and serves under `/_synapse/metrics`. If you do not wish your metrics be publicly exposed, you will need to either filter it out at your load balancer, or use the second method.

The second method runs the metrics server on a different port, in a different thread to Synapse. This can make it more resilient to heavy load meaning metrics cannot be retrieved, and can be exposed to just internal networks easier. The served metrics are available over HTTP only, and will be available at `/_synapse/metrics`.

Add a new listener to `homeserver.yaml` as in this example:

```

listeners:
  - port: 8008
    tls: false
    type: http
    x_forwarded: true
    bind_addresses: ['::1', '127.0.0.1']

resources:
  - names: [client, federation]
    compress: false

# beginning of the new metrics listener
- port: 9000
  type: metrics
  bind_addresses: ['::1', '127.0.0.1']

```

4. Restart Synapse.

5. Add a Prometheus target for Synapse.

It needs to set the `metrics_path` to a non-default value (under `scrape_configs`):

```

- job_name: "synapse"
  scrape_interval: 15s
  metrics_path: "/_synapse/metrics"
  static_configs:
    - targets: ["my.server.here:port"]

```

where `my.server.here` is the IP address of Synapse, and `port` is the listener port configured with the `metrics` resource.

If your prometheus is older than 1.5.2, you will need to replace `static_configs` in the above with `target_groups`.

6. Restart Prometheus.

7. Consider using the [grafana dashboard](#) and required [recording rules](#)

Monitoring workers

To monitor a Synapse installation using [workers](#), every worker needs to be monitored independently, in addition to the main homeserver process. This is because workers don't send their metrics to the main homeserver process, but expose them directly (if they are configured to do so).

To allow collecting metrics from a worker, you need to add a `metrics` listener to its configuration, by adding the following under `worker_listeners`:

```
- type: metrics
  bind_address: ''
  port: 9101
```

The `bind_address` and `port` parameters should be set so that the resulting listener can be reached by prometheus, and they don't clash with an existing worker. With this example, the worker's metrics would then be available on `http://127.0.0.1:9101`.

Example Prometheus target for Synapse with workers:

```
- job_name: "synapse"
  scrape_interval: 15s
  metrics_path: "/_synapse/metrics"
  static_configs:
    - targets: ["my.server.here:port"]
      labels:
        instance: "my.server"
        job: "master"
        index: 1
    - targets: ["my.workerserver.here:port"]
      labels:
        instance: "my.server"
        job: "generic_worker"
        index: 1
    - targets: ["my.workerserver.here:port"]
      labels:
        instance: "my.server"
        job: "generic_worker"
        index: 2
    - targets: ["my.workerserver.here:port"]
      labels:
        instance: "my.server"
        job: "media_repository"
        index: 1
```

Labels (`instance`, `job`, `index`) can be defined as anything. The labels are used to group graphs in grafana.

Renaming of metrics & deprecation of old names in 1.2

Synapse 1.2 updates the Prometheus metrics to match the naming convention of the upstream `prometheus_client`. The old names are considered deprecated and will be removed in a future version of Synapse. **The old names will be disabled by default in Synapse v1.71.0 and removed altogether in Synapse v1.73.0.**

New Name	
python_gc_objects_collected_total	python
python_gc_objects_uncollectable_total	python
python_gc_collections_total	python
process_cpu_seconds_total	process
synapse_federation_client_sent_transactions_total	syna
synapse_federation_client_events_processed_total	syna
synapse_event_processing_loop_count_total	syna
synapse_event_processing_loop_room_count_total	syna
synapse_util_caches_cache_hits	syna
synapse_util_caches_cache_size	syna
synapse_util_caches_cache_evicted_size	syna
synapse_util_caches_cache	syna
synapse_util_caches_response_cache_size	syna
synapse_util_caches_response_cache_hits	syna
synapse_util_caches_response_cache_evicted_size	syna
synapse_util_metrics_block_count_total	syna
synapse_util_metrics_block_time_seconds_total	syna
synapse_util_metrics_block_ru_utime_seconds_total	syna
synapse_util_metrics_block_ru_stime_seconds_total	syna
synapse_util_metrics_block_db_txn_count_total	syna
synapse_util_metrics_block_db_txn_duration_seconds_total	syna
synapse_util_metrics_block_db_sched_duration_seconds_total	syna
synapse_background_process_start_count_total	syna
synapse_background_process_ru_utime_seconds_total	syna
synapse_background_process_ru_stime_seconds_total	syna
synapse_background_process_db_txn_count_total	syna
synapse_background_process_db_txn_duration_seconds_total	syna
synapse_background_process_db_sched_duration_seconds_total	syna
synapse_storage_events_persisted_events_total	syna
synapse_storage_events_persisted_events_sep_total	syna
synapse_storage_events_state_delta_total	syna
synapse_storage_events_state_delta_single_event_total	syna
synapse_storage_events_state_delta_reuse_delta_total	syna
synapse_federation_server_received_pdus_total	syna
synapse_federation_server_received_edus_total	syna

New Name	
synapse_handler_presence_notified_presence_total	syna
synapse_handler_presence_federation_presence_out_total	syna
synapse_handler_presence_presence_updates_total	syna
synapse_handler_presence_timers_fired_total	syna
synapse_handler_presence_federation_presence_total	syna
synapse_handler_presence_bump_active_time_total	syna
synapse_federation_client_sent_edus_total	syna
synapse_federation_client_sent_pdu_destinations_count_total	syna
synapse_federation_client_sent_pdu_destinations_total	syna
synapse_handlers_appservice_events_processed_total	syna
synapse_notifier_notified_events_total	syna
synapse_push_bulk_push_rule_evaluator_push_rules_invalidation_counter_total	syna
synapse_push_bulk_push_rule_evaluator_push_rules_state_size_counter_total	syna
synapse_http_httppusher_http_pushes_processed_total	syna
synapse_http_httppusher_http_pushes_failed_total	syna
synapse_http_httppusher_badge_updates_processed_total	syna
synapse_http_httppusher_badge_updates_failed_total	syna
synapse_admin_mau_current	syna
synapse_admin_mau_max	syna
synapse_admin_mau_registered_reserved_users	syna

Removal of deprecated metrics & time based counters becoming histograms in 0.31.0

The duplicated metrics deprecated in Synapse 0.27.0 have been removed.

All time duration-based metrics have been changed to be seconds. This affects:

msec -> sec metrics
python_gc_time
python_twisted_reactor_tick_time
synapse_storage_query_time
synapse_storage_schedule_time
synapse_storage_transaction_time

Several metrics have been changed to be histograms, which sort entries into buckets and allow better analysis. The following metrics are now histograms:

Altered metrics
python_gc_time
python_twisted_reactor_pending_calls
python_twisted_reactor_tick_time
synapse_http_server_response_time_seconds
synapse_storage_query_time
synapse_storage_schedule_time
synapse_storage_transaction_time

Block and response metrics renamed for 0.27.0

Synapse 0.27.0 begins the process of rationalising the duplicate `*:count` metrics reported for the resource tracking for code blocks and HTTP requests.

At the same time, the corresponding `*:total` metrics are being renamed, as the `:total` suffix no longer makes sense in the absence of a corresponding `:count` metric.

To enable a graceful migration path, this release just adds new names for the metrics being renamed. A future release will remove the old ones.

The following table shows the new metrics, and the old metrics which they are replacing.

New name	Old name
synapse_util_metrics_block_count	synapse_util_metrics_block
synapse_util_metrics_block_time_seconds	synapse_util_metrics_block
synapse_util_metrics_block_ru_utime_seconds	synapse_util_metrics_block
synapse_util_metrics_block_ru_stime_seconds	synapse_util_metrics_block
synapse_util_metrics_block_db_txn_count	synapse_util_metrics_block
synapse_util_metrics_block_db_txn_duration_seconds	synapse_util_metrics_block
synapse_http_server_response_count	synapse_http_server_request_count
synapse_http_server_response_count	synapse_http_server_response_count
synapse_http_server_response_count	synapse_http_server_response_count

New name	Old name
synapse_http_server_response_count	synapse_http_server_respo
synapse_http_server_response_count	synapse_http_server_respo
synapse_http_server_response_count	synapse_http_server_respo
synapse_http_server_response_time_seconds	synapse_http_server_respo
synapse_http_server_response_ru_utime_seconds	synapse_http_server_respo
synapse_http_server_response_ru_stime_seconds	synapse_http_server_respo
synapse_http_server_response_db_txn_count	synapse_http_server_respo
synapse_http_server_response_db_txn_duration_seconds	synapse_http_server_respo

Standard Metric Names

As of synapse version 0.18.2, the format of the process-wide metrics has been changed to fit prometheus standard naming conventions. Additionally the units have been changed to seconds, from milliseconds.

New name	Old name
process_cpu_user_seconds_total	process_resource_utime / 1000
process_cpu_system_seconds_total	process_resource_stime / 1000
process_open_fds (no 'type' label)	process_fds

The python-specific counts of garbage collector performance have been renamed.

New name	Old name
python_gc_time	reactor_gc_time
python_gc_unreachable_total	reactor_gc_unreachable
python_gc_counts	reactor_gc_counts

The twisted-specific reactor metrics have been renamed.

New name	Old name
python_twisted_reactor_pending_calls	reactor_pending_calls
python_twisted_reactor_tick_time	reactor_tick_time

Reporting Homeserver Usage Statistics

When generating your Synapse configuration file, you are asked whether you would like to report usage statistics to Matrix.org. These statistics provide the foundation a glimpse into the number of Synapse homeservers participating in the network, as well as statistics such as the number of rooms being created and messages being sent. This feature is sometimes affectionately called "phone home" stats. Reporting [is optional](#) and the reporting endpoint [can be configured](#), in case you would like to instead report statistics from a set of homeservers to your own infrastructure.

This documentation aims to define the statistics available and the homeserver configuration options that exist to tweak it.

Available Statistics

The following statistics are sent to the configured reporting endpoint:

Statistic Name	Type	Description
<code>homeserver</code>	string	The homeserver's server name.
<code>memory_rss</code>	int	The memory usage of the process (in kilobytes on Unix-based systems, bytes on MacOS).
<code>cpu_average</code>	int	CPU time in % of a single core (not % of all cores).
<code>server_context</code>	string	An arbitrary string used to group statistics from a set of homeservers.
<code>timestamp</code>	int	The current time, represented as the number of seconds since the epoch.
<code>uptime_seconds</code>	int	The number of seconds since the homeserver was last started.
<code>python_version</code>	string	The Python version number in use (e.g "3.7.1"). Taken from <code>sys.version_info</code> .
<code>total_users</code>	int	The number of registered users on the homeserver.
<code>total_nonbridged_users</code>	int	The number of users, excluding those created by an Application Service.
<code>daily_user_type_native</code>	int	The number of native users created in the last 24 hours.

Statistic Name	Type	Description
<code>daily_user_type_guest</code>	int	The number of guest users created in the last 24 hours.
<code>daily_user_type_bridged</code>	int	The number of users created by Application Services in the last 24 hours.
<code>total_room_count</code>	int	The total number of rooms present on the homeserver.
<code>daily_active_users</code>	int	The number of unique users ¹ that have used the homeserver in the last 24 hours.
<code>monthly_active_users</code>	int	The number of unique users ¹ that have used the homeserver in the last 30 days.
<code>daily_active_rooms</code>	int	The number of rooms that have had a (state) event with the type <code>m.room.message</code> sent in them in the last 24 hours.
<code>daily_active_e2ee_rooms</code>	int	The number of rooms that have had a (state) event with the type <code>m.room.encrypted</code> sent in them in the last 24 hours.
<code>daily_messages</code>	int	The number of (state) events with the type <code>m.room.message</code> seen in the last 24 hours.
<code>daily_e2ee_messages</code>	int	The number of (state) events with the type <code>m.room.encrypted</code> seen in the last 24 hours.
<code>daily_sent_messages</code>	int	The number of (state) events sent by a local user with the type <code>m.room.message</code> seen in the last 24 hours.
<code>daily_sent_e2ee_messages</code>	int	The number of (state) events sent by a local user with the type <code>m.room.encrypted</code> seen in the last 24 hours.
<code>r30v2_users_all</code>	int	The number of 30 day retained users, with a revised algorithm. Defined as users that appear more than once in the past 60 days, and have more than 30 days between the most and least recent appearances in the past 60 days. Includes clients that do not fit into the below r30 client types.
<code>r30v2_users_android</code>	int	The number of 30 day retained users, as defined above. Filtered only to clients

Statistic Name	Type	Description
		with ("riot" or "element") and "android" (case-insensitive) in the user agent string.
<code>r30v2_users_ios</code>	int	The number of 30 day retained users, as defined above. Filtered only to clients with ("riot" or "element") and "ios" (case-insensitive) in the user agent string.
<code>r30v2_users_electron</code>	int	The number of 30 day retained users, as defined above. Filtered only to clients with ("riot" or "element") and "electron" (case-insensitive) in the user agent string.
<code>r30v2_users_web</code>	int	The number of 30 day retained users, as defined above. Filtered only to clients with "mozilla" or "gecko" (case-insensitive) in the user agent string.
<code>cache_factor</code>	int	The configured <code>global_factor</code> value for caching.
<code>event_cache_size</code>	int	The configured <code>event_cache_size</code> value for caching.
<code>database_engine</code>	string	The database engine that is in use. Either "psycopg2" meaning PostgreSQL is in use, or "sqlite3" for SQLite3.
<code>database_server_version</code>	string	The version of the database server. Examples being "10.10" for PostgreSQL server version 10.0, and "3.38.5" for SQLite 3.38.5 installed on the system.
<code>log_level</code>	string	The log level in use. Examples are "INFO", "WARNING", "ERROR", "DEBUG", etc.

¹ Native matrix users and guests are always counted. If the `track_puppeted_user_ips` option is set to `true`, "puppeted" users (users that an Application Service have performed an action on behalf of) will also be counted. Note that an Application Service can "puppet" any user in their `user namespace`, not only users that the Application Service has created. If this happens, the Application Service will additionally be counted as a user (irrespective of `track_puppeted_user_ips`).

Using a Custom Statistics Collection Server

If statistics reporting is enabled, the endpoint that Synapse sends metrics to is configured by the `report_stats_endpoint` config option. By default, statistics are sent to Matrix.org.

If you would like to set up your own statistics collection server and send metrics there, you may consider using one of the following known implementations:

- [Matrix.org's Panopticon](#)
- [Famedly's Barad-dûr](#)
- [Synapse Usage Exporter for Prometheus](#)

Monthly Active Users

Synapse can be configured to record the number of monthly active users (also referred to as MAU) on a given homeserver. For clarity's sake, MAU only tracks local users.

Please note that the metrics recorded by the [Homeserver Usage Stats](#) are calculated differently. The `monthly_active_users` from the usage stats does not take into account any of the rules below, and counts any users who have made a request to the homeserver in the last 30 days.

See the [configuration manual](#) for details on how to configure MAU.

Calculating active users

Individual user activity is measured in active days. If a user performs an action, the exact time of that action is then recorded. When calculating the MAU figure, any users with a recorded action in the last 30 days are considered part of the cohort. Days are measured as a rolling window from the current system time to 30 days ago.

So for example, if Synapse were to calculate the active users on the 15th July at 13:25, it would include any activity from 15th June 13:25 onwards.

A user is **never** considered active if they are either:

- Part of the trial day cohort (described below)
- Owned by an application service
 - Note: This **only** covers users that are part of an application service `namespaces.users` registration. The namespace must also be marked as `exclusive`.

Otherwise, any request to Synapse will mark the user as active. Please note that registration will not mark a user as active *unless* they register with a 3pid that is included in the config field `mau_limits_reserved_threepids`.

The Prometheus metric for MAU is refreshed every 5 minutes.

Once an hour, Synapse checks to see if any users are inactive (with only activity timestamps later than 30 days). These users are removed from the active users cohort. If they then become active, they are immediately restored to the cohort.

It is important to note that **deactivated** users are not immediately removed from the pool of active users, but as these users won't perform actions they will eventually be removed from the cohort.

Trial days

If the config option `mau_trial_days` is set, a user must have been active this many days **after** registration to be active. A user is in the trial period if their registration timestamp (also known as the `creation_ts`) is less than `mau_trial_days` old.

As an example, if `mau_trial_days` is set to `3` and a user is active **after** 3 days (72 hours from registration time) then they will be counted as active.

The `mau_appservice_trial_days` config further extends this rule by applying different durations depending on the `appservice_id` of the user. Users registered by an application service will be recorded with an `appservice_id` matching the `id` key in the registration file for that service.

Limiting usage of the homeserver when the maximum MAU is reached

If both config options `limit_usage_by_mau` and `max_mau_value` is set, and the current MAU value exceeds the maximum value, the homeserver will begin to block some actions.

Individual users matching **any** of the below criteria never have their actions blocked:

- Considered part of the cohort of MAU users.
- Considered part of the trial period.
- Registered as a `support` user.
- Application service users if `track_appservice_user_ips` is NOT set.

Please note that server admins are **not** exempt from blocking.

The following actions are blocked when the MAU limit is exceeded:

- Logging in
- Sending events
- Creating rooms
- Syncing

Registration is also blocked for all new signups *unless* the user is registering with a threepid included in the `mau_limits_reserved_threepids` config value.

When a request is blocked, the response will have the `errcode` `M_RESOURCE_LIMIT_EXCEEDED`.

Metrics

Synapse records several different prometheus metrics for MAU.

`synapse_admin_mau_current` records the current MAU figure for native (non-application-service) users.

`synapse_admin_mau_max` records the maximum MAU as dictated by the `max_mau_value` config value.

`synapse_admin_mau_current_mau_by_service` records the current MAU including application service users. The label `app_service` can be used to filter by a specific service ID. This *also* includes non-application-service users under `app_service=native`.

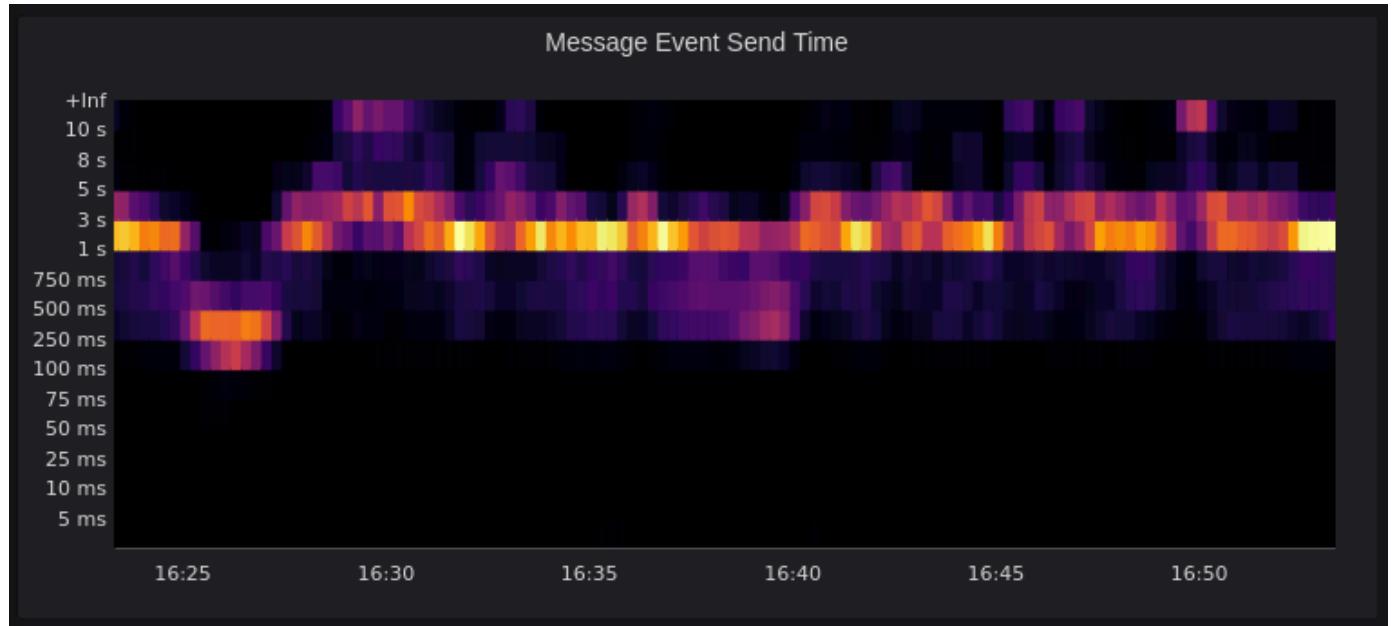
`synapse_admin_mau_registered_reserved_users` records the number of users specified in `mau_limits_reserved_threepids` which have registered accounts on the homeserver.

Understanding Synapse through Grafana graphs

It is possible to monitor much of the internal state of Synapse using [Prometheus](#) metrics and [Grafana](#). A guide for configuring Synapse to provide metrics is available [here](#) and information on setting up Grafana is [here](#). In this setup, Prometheus will periodically scrape the information Synapse provides and store a record of it over time. Grafana is then used as an interface to query and present this information through a series of pretty graphs.

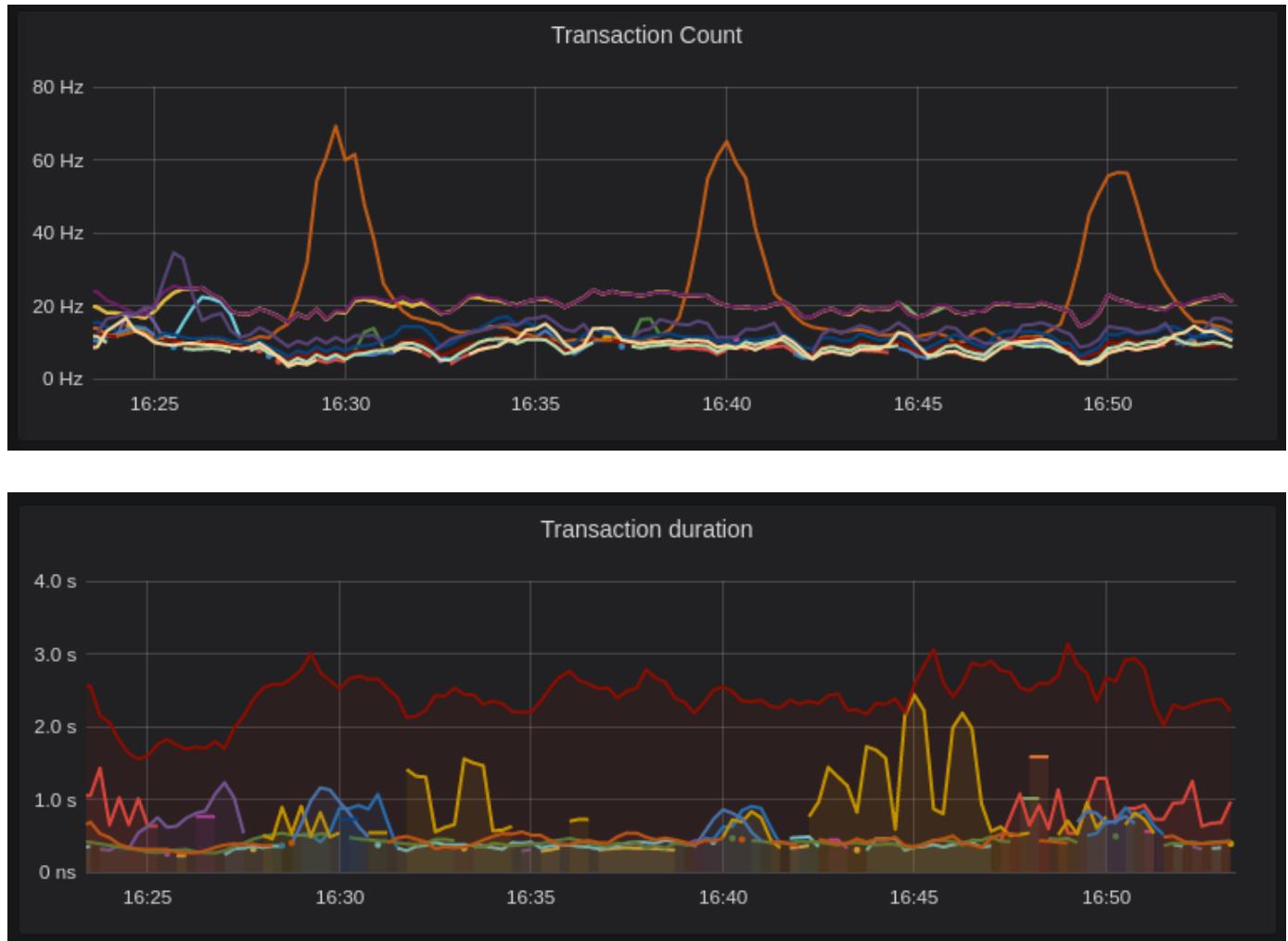
Once you have grafana set up, and assuming you're using [our grafana dashboard template](#), look for the following graphs when debugging a slow/overloaded Synapse:

Message Event Send Time

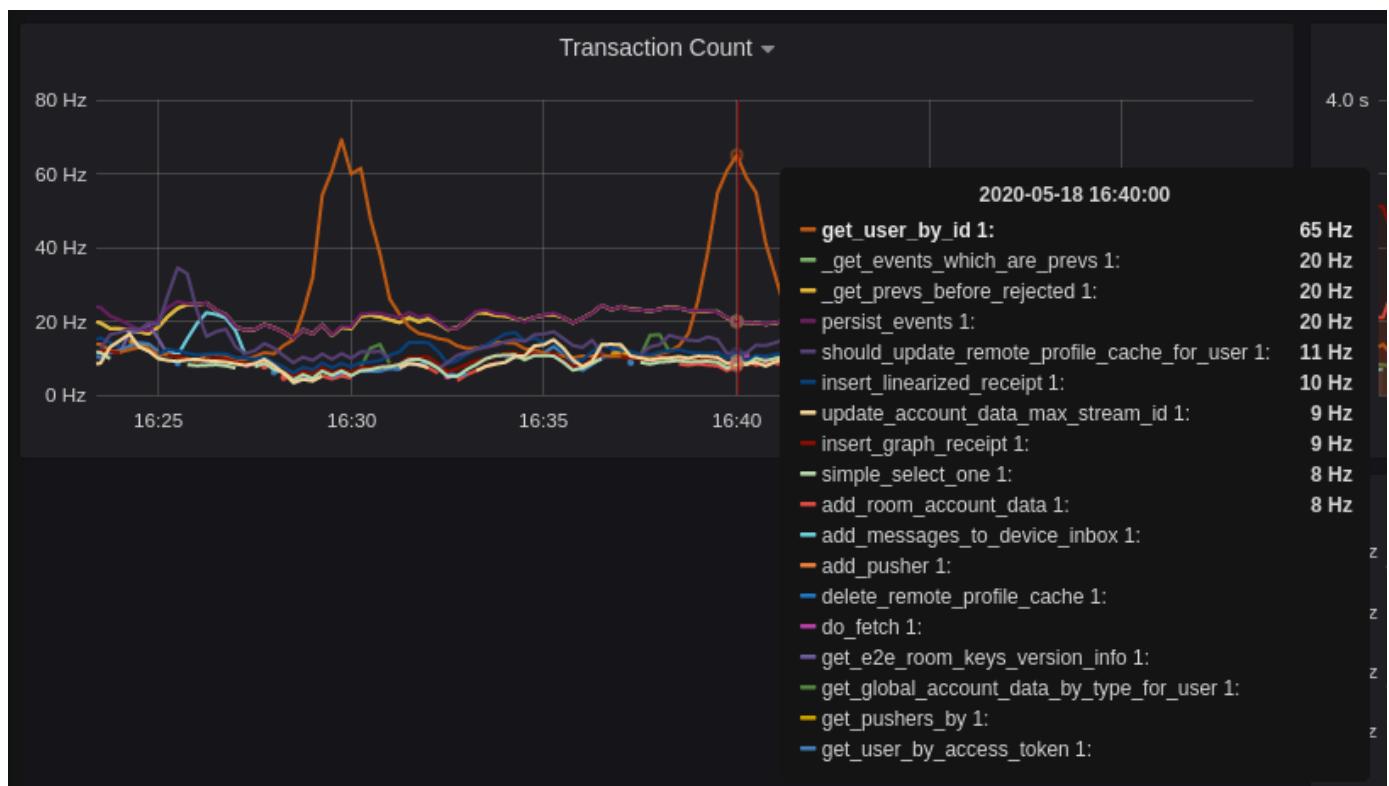


This, along with the CPU and Memory graphs, is a good way to check the general health of your Synapse instance. It represents how long it takes for a user on your homeserver to send a message.

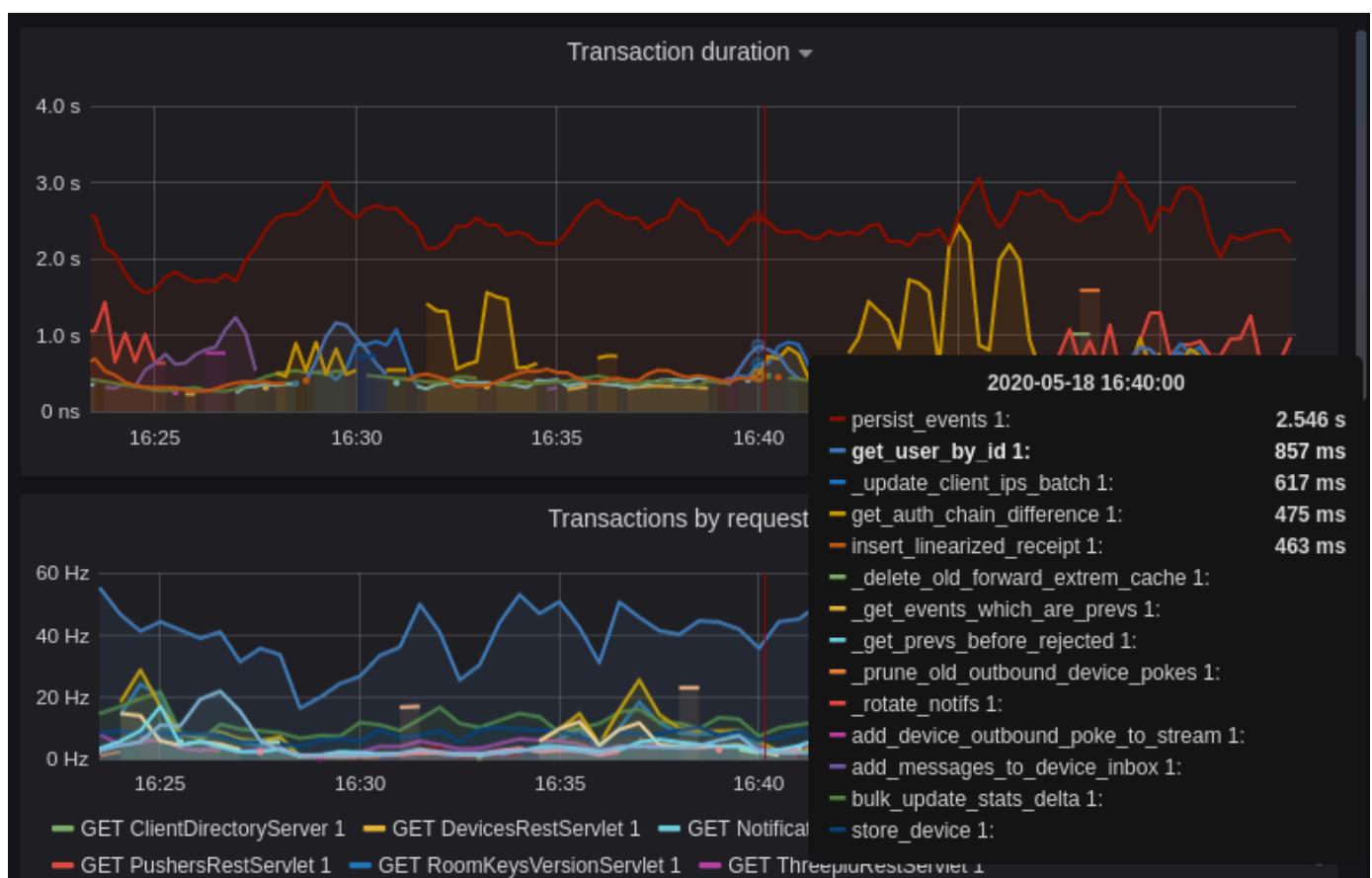
Transaction Count and Transaction Duration



These graphs show the database transactions that are occurring the most frequently, as well as those that are taking the most amount of time to execute.



In the first graph, we can see obvious spikes corresponding to lots of `get_user_by_id` transactions. This would be useful information to figure out which part of the Synapse codebase is potentially creating a heavy load on the system. However, be sure to cross-reference this with Transaction Duration, which states that `get_users_by_id` is actually a very quick database transaction and isn't causing as much load as others, like `persist_events`:



Still, it's probably worth investigating why we're getting users from the database that often, and whether it's possible to reduce the amount of queries we make by adjusting our cache factor(s).

The `persist_events` transaction is responsible for saving new room events to the Synapse database, so can often show a high transaction duration.

Federation

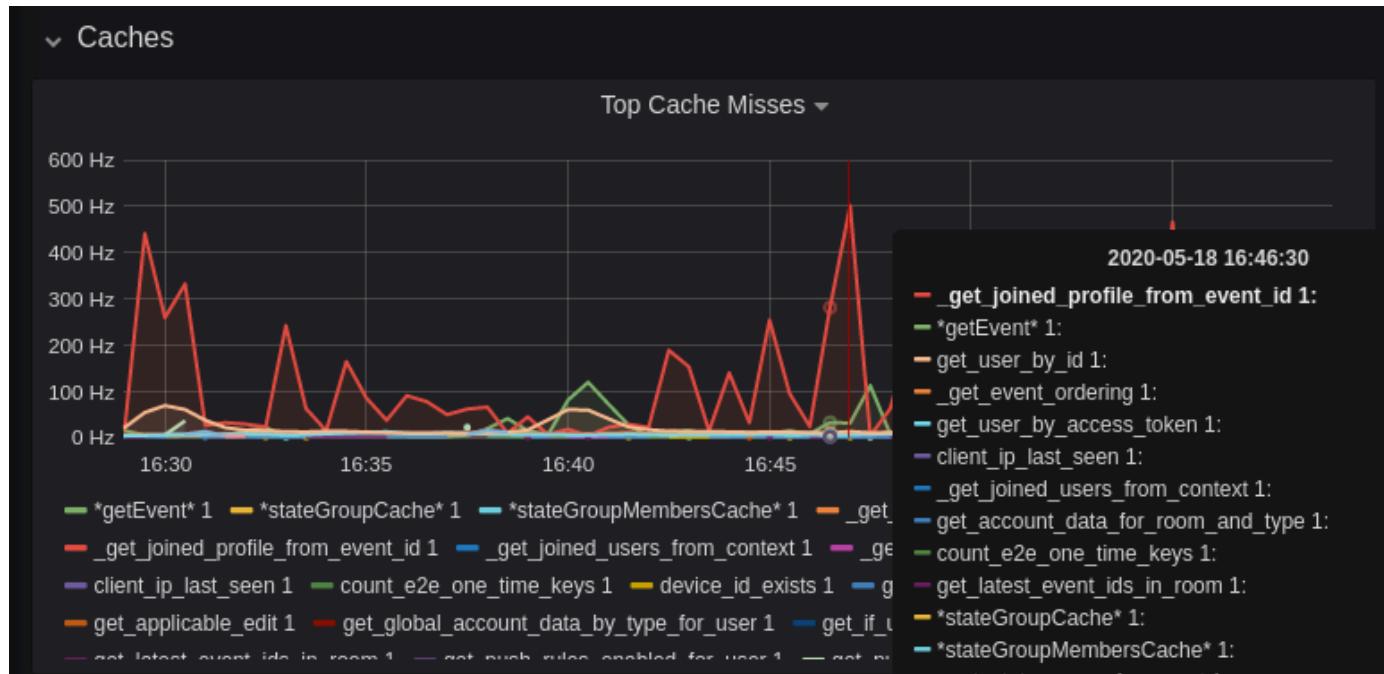
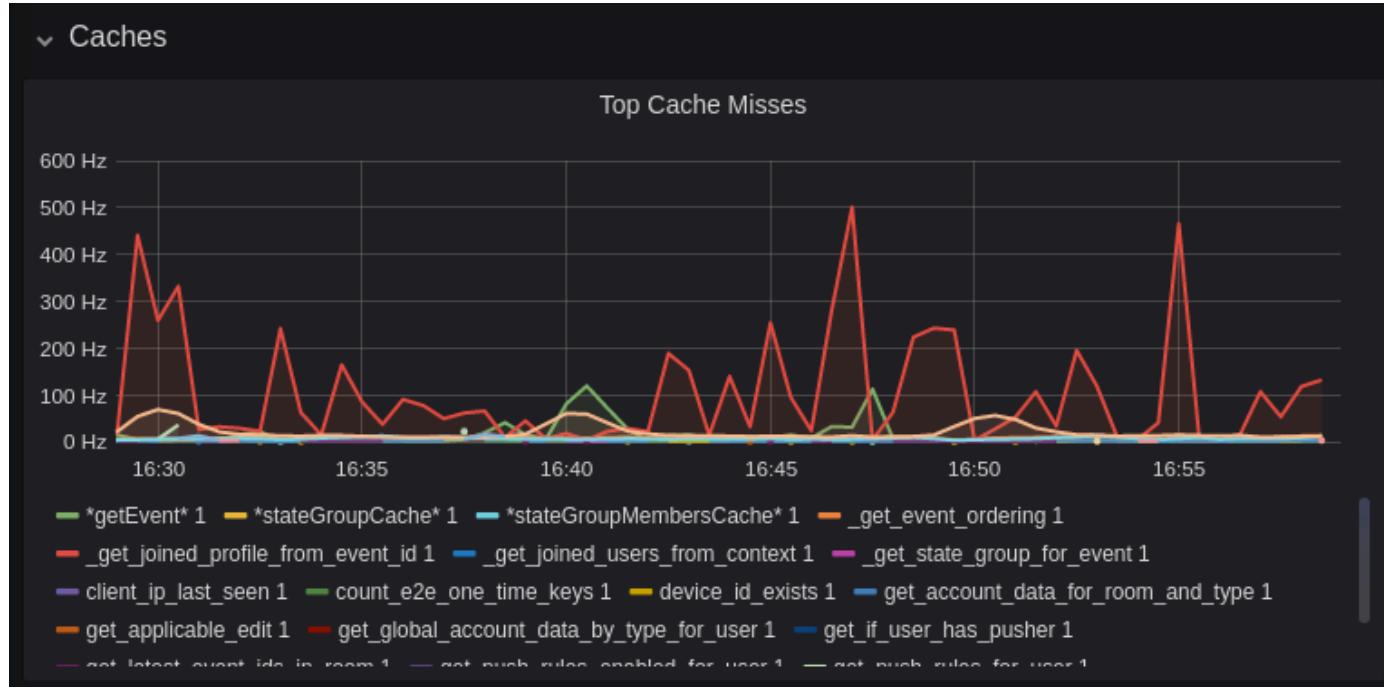
The charts in the "Federation" section show information about incoming and outgoing federation requests. Federation data can be divided into two basic types:

- PDU (Persistent Data Unit) - room events: messages, state events (join/leave), etc. These are permanently stored in the database.
- EDU (Ephemeral Data Unit) - other data, which need not be stored permanently, such as read receipts, typing notifications.

The "Outgoing EDUs by type" chart shows the EDUs within outgoing federation requests by type: `m.device_list_update`, `m.direct_to_device`, `m.presence`, `m.receipt`, `m.typing`.

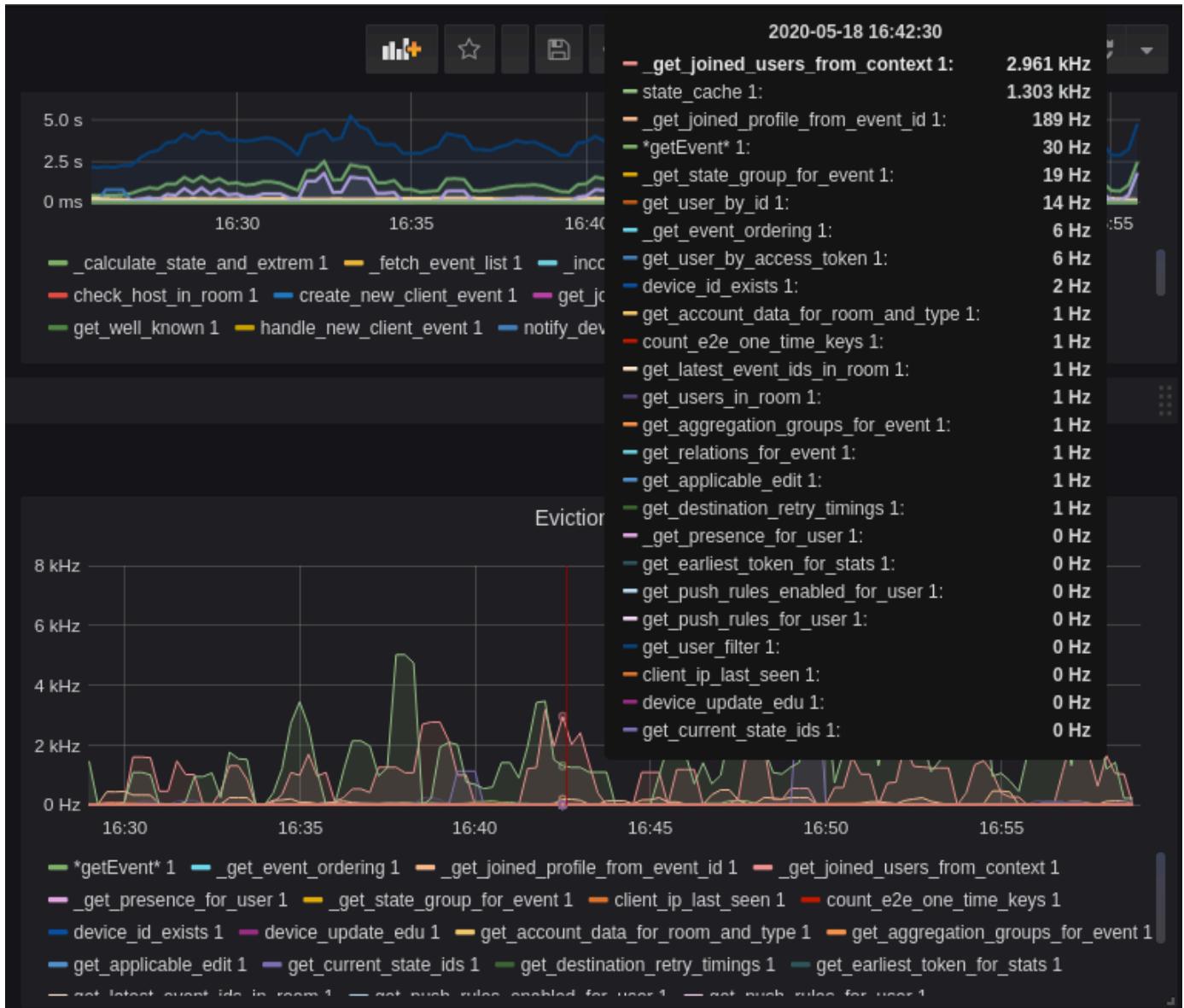
If you see a large number of `m.presence` EDUs and are having trouble with too much CPU load, you can disable `presence` in the Synapse config. See also [#3971](#).

Caches



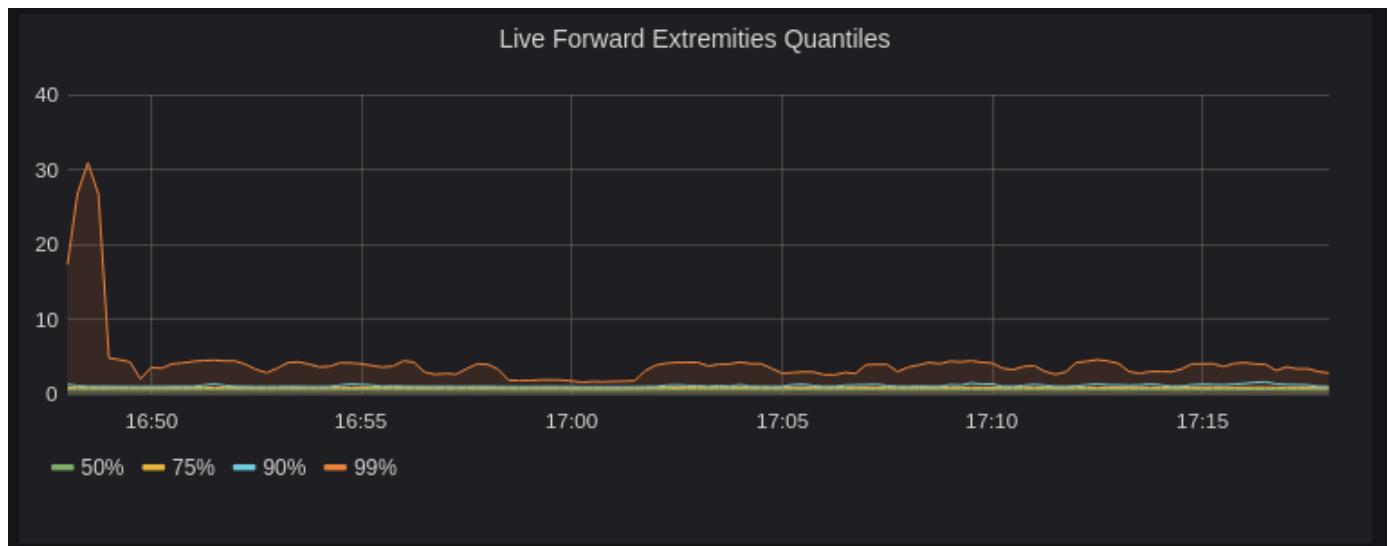
This is quite a useful graph. It shows how many times Synapse attempts to retrieve a piece of data from a cache which the cache did not contain, thus resulting in a call to the database. We can see here that the `_get_joined_profile_from_event_id` cache is being requested a lot, and often the data we're after is not cached.

Cross-referencing this with the Eviction Rate graph, which shows that entries are being evicted from `_get_joined_profile_from_event_id` quite often:



we should probably consider raising the size of that cache by raising its cache factor (a multiplier value for the size of an individual cache). Information on doing so is available [here](#) (note that the configuration of individual cache factors through the configuration file is available in Synapse v1.14.0+, whereas doing so through environment variables has been supported for a very long time). Note that this will increase Synapse's overall memory usage.

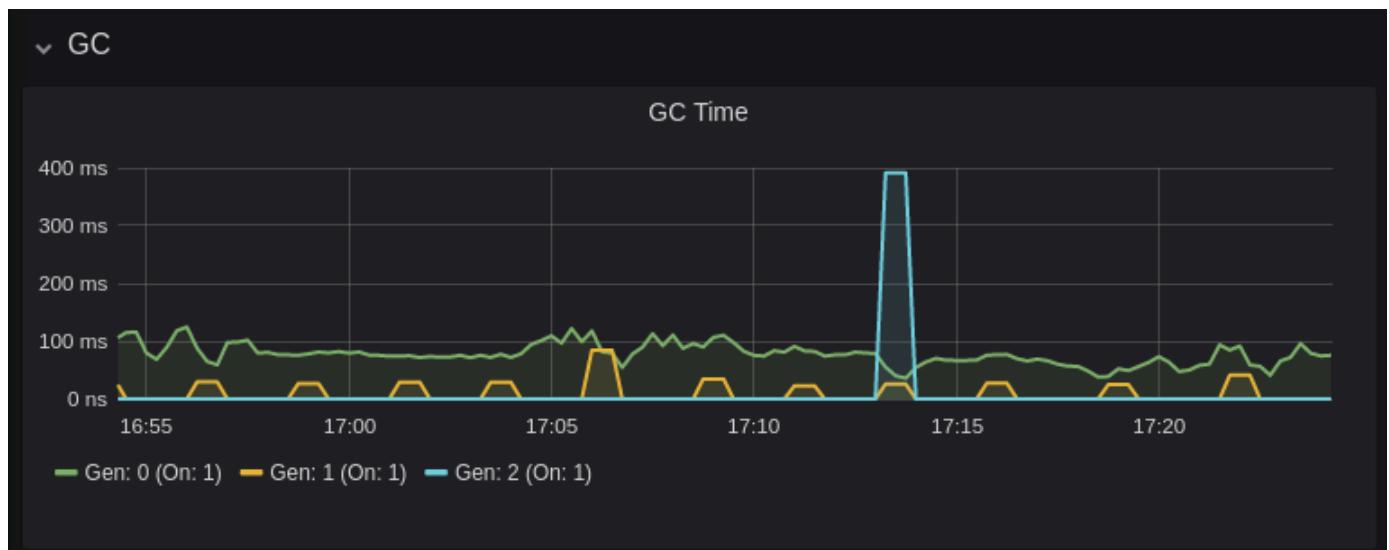
Forward Extremities



Forward extremities are the leaf events at the end of a DAG in a room, aka events that have no children. The more that exist in a room, the more [state resolution](#) that Synapse needs to perform (hint: it's an expensive operation). While Synapse has code to prevent too many of these existing at one time in a room, bugs can sometimes make them crop up again.

If a room has >10 forward extremities, it's worth checking which room is the culprit and potentially removing them using the SQL queries mentioned in [#1760](#).

Garbage Collection



Large spikes in garbage collection times (bigger than shown here, I'm talking in the multiple seconds range), can cause lots of problems in Synapse performance. It's more an indicator of problems, and a symptom of other problems though, so check other graphs for what might be causing it.

Final Thoughts

If you're still having performance problems with your Synapse instance and you've tried everything you can, it may just be a lack of system resources. Consider adding more CPU and RAM, and make use of [worker mode](#) to make use of multiple CPU cores / multiple machines for your homeserver.

Some useful SQL queries for Synapse Admins

Size of full matrix db

```
SELECT pg_size.pretty( pg_database_size( 'matrix' ) );
```

Result example:

```
pg_size.pretty
-----
 6420 MB
(1 row)
```

Show top 20 larger tables by row count

```
SELECT relname, n_live_tup AS "rows"
  FROM pg_stat_user_tables
 ORDER BY n_live_tup DESC
 LIMIT 20;
```

This query is quick, but may be very approximate, for exact number of rows use:

```
SELECT COUNT(*) FROM <table_name>;
```

Result example:

```
state_groups_state - 161687170
event_auth - 8584785
event_edges - 6995633
event_json - 6585916
event_reference_hashes - 6580990
events - 6578879
received_transactions - 5713989
event_to_state_groups - 4873377
stream_ordering_to_exterm - 4136285
current_state_delta_stream - 3770972
event_search - 3670521
state_events - 2845082
room_memberships - 2785854
cache_invalidation_stream - 2448218
state_groups - 1255467
state_group_edges - 1229849
current_state_events - 1222905
users_in_public_rooms - 364059
device_lists_stream - 326903
user_directory_search - 316433
```

Show top 20 larger tables by storage size

```
SELECT nspname || '.' || relname AS "relation",
       pg_size.pretty(pg_total_relation_size(c.oid)) AS "total_size"
  FROM pg_class c
 LEFT JOIN pg_namespace n ON (n.oid = c.relnamespace)
 WHERE nspname NOT IN ('pg_catalog', 'information_schema')
   AND c.relkind <> 'i'
   AND nspname !~ '^pg_toast'
 ORDER BY pg_total_relation_size(c.oid) DESC
 LIMIT 20;
```

Result example:

```
public.state_groups_state - 27 GB
public.event_json - 9855 MB
public.events - 3675 MB
public.event_edges - 3404 MB
public.received_transactions - 2745 MB
public.event_reference_hashes - 1864 MB
public.event_auth - 1775 MB
public.stream_ordering_to_exterm - 1663 MB
public.event_search - 1370 MB
public.room_memberships - 1050 MB
public.event_to_state_groups - 948 MB
public.current_state_delta_stream - 711 MB
public.state_events - 611 MB
public.presence_stream - 530 MB
public.current_state_events - 525 MB
public.cache_invalidation_stream - 466 MB
public.receipts_linearized - 279 MB
public.state_groups - 160 MB
public.device_lists_remote_cache - 124 MB
public.state_group_edges - 122 MB
```

Show top 20 larger rooms by state events count

You get the same information when you use the [admin API](#) and set parameter `order_by=state_events`.

```
SELECT r.name, s.room_id, s.current_state_events
  FROM room_stats_current s
  LEFT JOIN room_stats_state r USING (room_id)
 ORDER BY current_state_events DESC
LIMIT 20;
```

and by state_group_events count:

```
SELECT rss.name, s.room_id, COUNT(s.room_id)
  FROM state_groups_state s
  LEFT JOIN room_stats_state rss USING (room_id)
 GROUP BY s.room_id, rss.name
 ORDER BY COUNT(s.room_id) DESC
LIMIT 20;
```

plus same, but with join removed for performance reasons:

```
SELECT s.room_id, COUNT(s.room_id)
  FROM state_groups_state s
 GROUP BY s.room_id
 ORDER BY COUNT(s.room_id) DESC
LIMIT 20;
```

Show top 20 rooms by new events count in last 1 day:

```
SELECT e.room_id, r.name, COUNT(e.event_id) cnt
  FROM events e
  LEFT JOIN room_stats_state r USING (room_id)
 WHERE e.origin_server_ts >= DATE_PART('epoch', NOW() - INTERVAL '1 day') *
1000
  GROUP BY e.room_id, r.name
  ORDER BY cnt DESC
  LIMIT 20;
```

Show top 20 users on homeserver by sent events (messages) at last month:

Caution. This query does not use any indexes, can be slow and create load on the database.

```
SELECT COUNT(*), sender
  FROM events
 WHERE (type = 'm.room.encrypted' OR type = 'm.room.message')
   AND origin_server_ts >= DATE_PART('epoch', NOW() - INTERVAL '1 month') *
1000
  GROUP BY sender
  ORDER BY COUNT(*) DESC
  LIMIT 20;
```

Show last 100 messages from needed user, with room names:

```
SELECT e.room_id, r.name, e.event_id, e.type, e.content, j.json
  FROM events e
  LEFT JOIN event_json j USING (room_id)
  LEFT JOIN room_stats_state r USING (room_id)
 WHERE sender = '@LOGIN:example.com'
   AND e.type = 'm.room.message'
  ORDER BY stream_ordering DESC
  LIMIT 100;
```

Show rooms with names, sorted by events in this rooms

Sort and order with bash

```
echo "SELECT event_json.room_id, room_stats_state.name FROM event_json,
room_stats_state \
WHERE room_stats_state.room_id = event_json.room_id" | psql -d synapse -h
localhost -U synapse_user -t \
| sort | uniq -c | sort -n
```

Documentation for `psql` command line parameters:
<https://www.postgresql.org/docs/current/app-psql.html>

Sort and order with SQL

```
SELECT COUNT(*), event_json.room_id, room_stats_state.name
  FROM event_json, room_stats_state
 WHERE room_stats_state.room_id = event_json.room_id
 GROUP BY event_json.room_id, room_stats_state.name
 ORDER BY COUNT(*) DESC
 LIMIT 50;
```

Result example:

9459	!FPUfgzXYWTKgIrwKxW:matrix.org	This Week in Matrix
9459	!FPUfgzXYWTKgIrwKxW:matrix.org	This Week in Matrix
(TWIM)		
17799	!iDIOImbmXxwNngznsa:matrix.org	Linux in Russian
18739	!GnEEPYXUhoaHbkFBNX:matrix.org	Riot Android
23373	!QtykxKocfZaZOUrTwp:matrix.org	Matrix HQ
39504	!gTQfWzbYncrtNrvEkB:matrix.org	ru.[matrix]
43601	!iNmaIQExDMeqdITdHH:matrix.org	Riot
43601	!iNmaIQExDMeqdITdHH:matrix.org	Riot Web/Desktop

Lookup room state info by list of room_id

You get the same information when you use the [admin API](#).

```
SELECT rss.room_id, rss.name, rss.canonical_alias, rss.topic, rss.encryption,
       rsc.joined_members, rsc.local_users_in_room, rss.join_rules
  FROM room_stats_state rss
 LEFT JOIN room_stats_current rsc USING (room_id)
 WHERE room_id IN (
   '!OGEhHVWSdvArJzumhm:matrix.org',
   '!YTvKGNlinIzlkMTVRl:matrix.org'
 );
```

Show users and devices that have not been online for a while

```
SELECT user_id, device_id, user_agent, TO_TIMESTAMP(last_seen / 1000) AS
"last_seen"
FROM devices
WHERE last_seen < DATE_PART('epoch', NOW() - INTERVAL '3 month') * 1000;
```

Clear the cache of a remote user's device list

Forces the resync of a remote user's device list - if you have somehow cached a bad state, and the remote server is will not send out a device list update.

```
INSERT INTO device_lists_remote_resync
VALUES ('USER_ID', (EXTRACT(epoch FROM NOW()) * 1000)::BIGINT);
```

This [blog post by Jackson Chen](#) (Dec 2022) explains how to use many of the tools listed on this page. There is also an [earlier blog by Victor Berger](#) (June 2020), though this may be outdated in places.

List of useful tools and scripts for maintenance Synapse database:

Purge Remote Media API

The purge remote media API allows server admins to purge old cached remote media.

Purge Local Media API

This API deletes the *local* media from the disk of your own server.

Purge History API

The purge history API allows server admins to purge historic events from their database, reclaiming disk space.

synapse-compress-state

Tool for compressing (deduplicating) `state_groups_state` table.

SQL for analyzing Synapse PostgreSQL database stats

Some easy SQL that reports useful stats about your Synapse database.

How do State Groups work?

As a general rule, I encourage people who want to understand the deepest darkest secrets of the database schema to drop by [#synapse-dev:matrix.org](#) and ask questions.

However, one question that comes up frequently is that of how "state groups" work, and why the `state_groups_state` table gets so big, so here's an attempt to answer that question.

We need to be able to relatively quickly calculate the state of a room at any point in that room's history. In other words, we need to know the state of the room at each event in that room. This is done as follows:

A sequence of events where the state is the same are grouped together into a `state_group`; the mapping is recorded in `event_to_state_groups`. (Technically speaking, since a state event usually changes the state in the room, we are recording the state of the room *after* the given event id: which is to say, to a handwavey simplification, the first event in a state group is normally a state event, and others in the same state group are normally non-state-events.)

`state_groups` records, for each state group, the id of the room that we're looking at, and also the id of the first event in that group. (I'm not sure if that event id is used much in practice.)

Now, if we stored all the room state for each `state_group`, that would be a huge amount of data. Instead, for each state group, we normally store the difference between the state in that group and some other state group, and only occasionally (every 100 state changes or so) record the full state.

So, most state groups have an entry in `state_group_edges` (don't ask me why it's not a column in `state_groups`) which records the previous state group in the room, and `state_groups_state` records the differences in state since that previous state group.

A full state group just records the event id for each piece of state in the room at that point.

Known bugs with state groups

There are various reasons that we can end up creating many more state groups than we need: see <https://github.com/matrix-org/synapse/issues/3364> for more details.

Compression tool

There is a tool at <https://github.com/matrix-org/rust-synapse-compress-state> which can compress the `state_groups_state` on a room by-room basis (essentially, it reduces the number of "full" state groups). This can result in dramatic reductions of the storage used.

Request log format

HTTP request logs are written by synapse (see [synapse/http/site.py](#) for details).

See the following for how to decode the dense data available from the default logging configuration.

Part	Explanation
AAAA	Timestamp request was logged (not received)
BBBB	Logger name (<code>synapse.access.(http\ https).<tag></code> , where 'tag' is defined in the <code>listeners</code> config section, normally the port)
CCCC	Line number in code
DDDD	Log Level
EEEE	Request Identifier (This identifier is shared by related log lines)
FFFF	Source IP (Or X-Forwarded-For if enabled)
GGGG	Server Port
HHHH	Federated Server or Local User making request (blank if unauthenticated or not supplied). If this is of the form `@aaa:example.com
IIII	Total Time to process the request
JJJJ	Time to send response over network once generated (this may be negative if the socket is closed before the response is generated)
KKKK	Userland CPU time
LLLL	System CPU time
MMMM	Total time waiting for a free DB connection from the pool across all parallel DB work from this request
NNNN	Total time waiting for response to DB queries across all parallel DB work from this request
OOOO	Count of DB transactions performed
PPPP	Response body size

Part	Explanation
QQQQ	Response status code Suffixed with ! if the socket was closed before the response was generated. A 499! status code indicates that Synapse also cancelled request processing after the socket was closed.
RRRR	Request
SSSS	User-agent
TTTT	Events fetched from DB to service this request (note that this does not include events fetched from the cache)

MMMM / NNNN can be greater than IIII if there are multiple slow database queries running in parallel.

Some actions can result in multiple identical http requests, which will return the same data, but only the first request will report time/transactions in **KKKK / LLLL / MMMM / NNNN / 0000** - the others will be awaiting the first query to return a response and will simultaneously return with the first request, but with very small processing times.

Admin FAQ

How do I become a server admin?

If your server already has an admin account you should use the [User Admin API](#) to promote other accounts to become admins.

If you don't have any admin accounts yet you won't be able to use the admin API, so you'll have to edit the database manually. Manually editing the database is generally not recommended so once you have an admin account: use the admin APIs to make further changes.

```
UPDATE users SET admin = 1 WHERE name = '@foo:bar.com';
```

What servers are my server talking to?

Run this sql query on your db:

```
SELECT * FROM destinations;
```

What servers are currently participating in this room?

Run this sql query on your db:

```
SELECT DISTINCT split_part(state_key, ':', 2)
FROM current_state_events
WHERE room_id = '!cURbafjkfsMDVwdRDQ:matrix.org' AND membership = 'join';
```

What users are registered on my server?

```
SELECT NAME from users;
```

How can I export user data?

Synapse includes a Python command to export data for a specific user. It takes the homeserver configuration file and the full Matrix ID of the user to export:

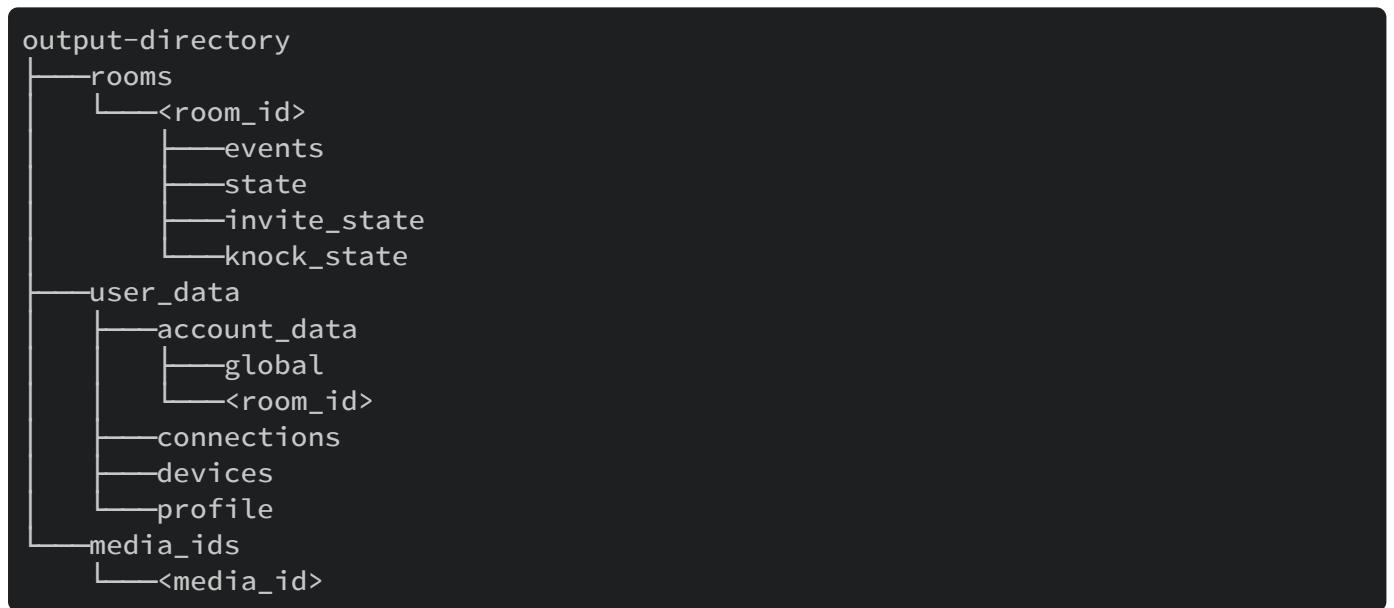
```
python -m synapse.app.admin_cmd -c <config_file> export-data <user_id> --output-directory <directory_path>
```

If you uses [Poetry](#) to run Synapse:

```
poetry run python -m synapse.app.admin_cmd -c <config_file> export-data <user_id> --output-directory <directory_path>
```

The directory to store the export data in can be customised with the `--output-directory` parameter; ensure that the provided directory is empty. If this parameter is not provided, Synapse defaults to creating a temporary directory (which starts with "synapse-exfiltrate") in `/tmp`, `/var/tmp`, or `/usr/tmp`, in that order.

The exported data has the following layout:



The `media_ids` folder contains only the metadata of the media uploaded by the user. It does not contain the media itself. Furthermore, only the `media_ids` that Synapse manages itself are exported. If another media repository (e.g. [matrix-media-repo](#)) is used, the data must be exported separately.

With the `media_ids` the media files can be downloaded. Media that have been sent in encrypted rooms are only retrieved in encrypted form. The following script can help with download the media files:

```

#!/usr/bin/env bash

# Parameters
#
#   source_directory: Directory which contains the export with the media_ids.
#   target_directory: Directory into which all files are to be downloaded.
#   repository_url: Address of the media repository resp. media worker.
#   serverName: Name of the server (`server_name` from homeserver.yaml).
#
# Example:
#       ./download_media.sh /tmp/export_data/media_ids/
#       /tmp/export_data/media_files/ http://localhost:8008 matrix.example.com

source_directory=$1
target_directory=$2
repository_url=$3
serverName=$4

mkdir -p $target_directory

for file in $source_directory/*; do
    filename=$(basename ${file})
    url=$repository_url/_matrix/media/v3/download/$serverName/$filename
    echo "Downloading $filename - $url"
    if ! wget -o /dev/null -P $target_directory $url; then
        echo "Could not download $filename"
    fi
done

```

How do I upgrade from a very old version of Synapse to the latest?

See [this](#) section in the upgrade docs.

Manually resetting passwords

Users can reset their password through their client. Alternatively, a server admin can reset a user's password using the [admin API](#).

I have a problem with my server. Can I just delete my database and start again?

Deleting your database is unlikely to make anything better.

It's easy to make the mistake of thinking that you can start again from a clean slate by dropping your database, but things don't work like that in a federated network: lots of other servers have information about your server.

For example: other servers might think that you are in a room, your server will think that you are not, and you'll probably be unable to interact with that room in a sensible way ever again.

In general, there are better solutions to any problem than dropping the database. Come and seek help in <https://matrix.to/#/#synapse:matrix.org>.

There are two exceptions when it might be sensible to delete your database and start again:

- You have *never* joined any rooms which are federated with other servers. For instance, a local deployment which the outside world can't talk to.
- You are changing the `server_name` in the homeserver configuration. In effect this makes your server a completely new one from the point of view of the network, so in this case it makes sense to start with a clean database. (In both cases you probably also want to clear out the `media_store`.)

I've stuffed up access to my room, how can I delete it to free up the alias?

Using the following curl command:

```
curl -H 'Authorization: Bearer <access-token>' -X DELETE
https://matrix.org/_matrix/client/r0/directory/room/<room-alias>
```

`<access-token>` - can be obtained in riot by looking in the riot settings, down the bottom is: Access Token:<click to reveal>

`<room-alias>` - the room alias, eg. #my_room:matrix.org this possibly needs to be URL encoded also, for example %23my_room%3Amatrix.org

How can I find the lines corresponding to a given HTTP request in my homeserver log?

Synapse tags each log line according to the HTTP request it is processing. When it finishes processing each request, it logs a line containing the words `Processed request: .` For example:

```
2019-02-14 22:35:08,196 - synapse.access.http.8008 - 302 - INFO - GET-37 - ::1 - 8008 - {@richvdh:localhost} Processed request: 0.173sec/0.001sec (0.002sec, 0.000sec) (0.027sec/0.026sec/2) 687B 200 "GET /_matrix/client/r0-sync HTTP/1.1" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36" [0 dbevts]"
```

Here we can see that the request has been tagged with `GET-37`. (The tag depends on the method of the HTTP request, so might start with `GET-`, `PUT-`, `POST-`, `OPTIONS-` or `DELETE-`.) So to find all lines corresponding to this request, we can do:

```
grep 'GET-37' homeserver.log
```

If you want to paste that output into a github issue or matrix room, please remember to surround it with triple-backticks (``` to make it legible (see [quoting code](#)).

What do all those fields in the 'Processed' line mean?

See [Request log format](#).

What are the biggest rooms on my server?

```
SELECT s.canonical_alias, g.room_id, count(*) AS num_rows
FROM
  state_groups_state AS g,
  room_stats_state AS s
WHERE g.room_id = s.room_id
GROUP BY s.canonical_alias, g.room_id
ORDER BY num_rows desc
LIMIT 10;
```

You can also use the [List Room API](#) and `order_by state_events`.

People can't accept room invitations from me

The typical failure mode here is that you send an invitation to someone to join a room or direct chat, but when they go to accept it, they get an error (typically along the lines of "Invalid signature"). They might see something like the following in their logs:

```
2019-09-11 19:32:04,271 - synapse.federation.transport.server - 288 - WARNING - GET-11752 - authenticate_request failed: 401: Invalid signature for server <server> with key ed25519:a_EqML: Unable to verify signature for <server>
```

This is normally caused by a misconfiguration in your reverse-proxy. See [the reverse proxy docs](#) and double-check that your settings are correct.

Help!! Synapse is slow and eats all my RAM/CPU!

First, ensure you are running the latest version of Synapse, using Python 3 with a [PostgreSQL database](#).

Synapse's architecture is quite RAM hungry currently - we deliberately cache a lot of recent room data and metadata in RAM in order to speed up common requests. We'll improve this in the future, but for now the easiest way to either reduce the RAM usage (at the risk of slowing things down) is to set the almost-undocumented `SYNAPSE_CACHE_FACTOR` environment variable. The default is 0.5, which can be decreased to reduce RAM usage in memory constrained environments, or increased if performance starts to degrade.

However, degraded performance due to a low cache factor, common on machines with slow disks, often leads to explosions in memory use due to backlogged requests. In this case, reducing the cache factor will make things worse. Instead, try increasing it drastically. 2.0 is a good starting value.

Using [libjemalloc](#) can also yield a significant improvement in overall memory use, and especially in terms of giving back RAM to the OS. To use it, the library must simply be put in the `LD_PRELOAD` environment variable when launching Synapse. On Debian, this can be done by installing the `libjemalloc2` package and adding this line to `/etc/default/matrix-synapse`:

```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libjemalloc.so.2
```

Note: You may need to set `PYTHONMALLOC=malloc` to ensure that `jemalloc` can accurately calculate memory usage. By default, Python uses its internal small-object allocator, which may interfere with jemalloc's ability to track memory consumption correctly. This could prevent the [cache_autotuning](#) feature from functioning as expected, as the Python allocator may not reach the memory threshold set by `max_cache_memory_usage`, thus not triggering the cache eviction process.

This made a significant difference on Python 2.7 - it's unclear how much of an improvement it provides on Python 3.x.

If you're encountering high CPU use by the Synapse process itself, you may be affected by a bug with presence tracking that leads to a massive excess of outgoing federation requests (see [discussion](#)). If metrics indicate that your server is also issuing far more outgoing federation requests than can be accounted for by your users' activity, this is a likely cause. The misbehavior can be worked around by disabling presence in the Synapse config file: [see here](#).

Running out of File Handles

If Synapse runs out of file handles, it typically fails badly - live-locking at 100% CPU, and/or failing to accept new TCP connections (blocking the connecting client). Matrix currently can legitimately use a lot of file handles, thanks to busy rooms like [#matrix:matrix.org](#) containing hundreds of participating servers. The first time a server talks in a room it will try to connect simultaneously to all participating servers, which could exhaust the available file descriptors between DNS queries & HTTPS sockets, especially if DNS is slow to respond. (We need to improve the routing algorithm used to be better than full mesh, but as of March 2019 this hasn't happened yet).

If you hit this failure mode, we recommend increasing the maximum number of open file handles to be at least 4096 (assuming a default of 1024 or 256). This is typically done by editing [/etc/security/limits.conf](#)

Separately, Synapse may leak file handles if inbound HTTP requests get stuck during processing - e.g. blocked behind a lock or talking to a remote server etc. This is best diagnosed by matching up the 'Received request' and 'Processed request' log lines and looking for any 'Processed request' lines which take more than a few seconds to execute. Please let us know at [#synapse:matrix.org](#) if you see this failure mode so we can help debug it, however.

Contributing

This document aims to get you started with contributing to Synapse!

1. Who can contribute to Synapse?

Everyone is welcome to contribute code to Synapse, provided that they are willing to license their contributions to Element under a [Contributor License Agreement](#) (CLA). This ensures that their contribution will be made available under an OSI-approved open-source license, currently Affero General Public License v3 (AGPLv3).

Please see the [Element blog post](#) for the full rationale.

2. What do I need?

If you are running Windows, the Windows Subsystem for Linux (WSL) is strongly recommended for development. More information about WSL can be found at <https://docs.microsoft.com/en-us/windows/wsl/install>. Running Synapse natively on Windows is not officially supported.

The code of Synapse is written in Python 3. To do pretty much anything, you'll need [a recent version of Python 3](#). Your Python also needs support for [virtual environments](#). This is usually built-in, but some Linux distributions like Debian and Ubuntu split it out into its own package. Running `sudo apt install python3-venv` should be enough.

A recent version of the Rust compiler is needed to build the native modules. The easiest way of installing the latest version is to use [rustup](#).

Synapse can connect to PostgreSQL via the [psycopg2](#) Python library. Building this library from source requires access to PostgreSQL's C header files. On Debian or Ubuntu Linux, these can be installed with `sudo apt install libpq-dev`.

Synapse has an optional, improved user search with better Unicode support. For that you need the development package of [libcudf](#). On Debian or Ubuntu Linux, this can be installed with `sudo apt install libcudf-dev`.

The source code of Synapse is hosted on GitHub. You will also need [a recent version of git](#).

For some tests, you will need [a recent version of Docker](#).

3. Get the source.

The preferred and easiest way to contribute changes is to fork the relevant project on GitHub, and then [create a pull request](#) to ask us to pull your changes into our repo.

Please base your changes on the `develop` branch.

```
git clone git@github.com:YOUR_GITHUB_USER_NAME/synapse.git
git checkout develop
```

If you need help getting started with git, this is beyond the scope of the document, but you can find many good git tutorials on the web.

4. Install the dependencies

Before installing the Python dependencies, make sure you have installed a recent version of Rust (see the "What do I need?" section above). The easiest way of installing the latest version is to use [rustup](#).

Synapse uses the [poetry](#) project to manage its dependencies and development environment. Once you have installed Python 3 and added the source, you should install `poetry`. Of their installation methods, we recommend [installing poetry using pipx](#),

```
pip install --user pipx
pipx install poetry
```

but see poetry's [installation instructions](#) for other installation methods.

Developing Synapse requires Poetry version 1.3.2 or later.

Next, open a terminal and install dependencies as follows:

```
cd path/where/you/have/cloned/the/repository
poetry install --extras all
```

This will install the runtime and developer dependencies for the project. Be sure to check that the `poetry install` step completed cleanly.

For OSX users, be sure to set `PKG_CONFIG_PATH` to support `icu4c`. Run `brew info icu4c` for more details.

Running Synapse via poetry

To start a local instance of Synapse in the locked poetry environment, create a config file:

```
cp docs/sample_config.yaml homeserver.yaml
cp docs/sample_log_config.yaml log_config.yaml
```

Now edit `homeserver.yaml`, things you might want to change include:

- Set a `server_name`
- Adjusting paths to be correct for your system like the `log_config` to point to the log config you just copied
- Using a [PostgreSQL database instead of SQLite](#)
- Adding a `registration_shared_secret` so you can use `register_new_matrix_user` command.

And then run Synapse with the following command:

```
poetry run python -m synapse.app.homeserver -c homeserver.yaml
```

If you get an error like the following:

```
importlib.metadata.PackageNotFoundError: matrix-synapse
```

this probably indicates that the `poetry install` step did not complete cleanly - go back and resolve any issues and re-run until successful.

5. Get in touch.

Join our developer community on Matrix: [#synapse-dev:matrix.org](#)!

6. Pick an issue.

Fix your favorite problem or perhaps find a [Good First Issue](#) to work on.

7. Turn coffee into code and documentation!

There is a growing amount of documentation located in the `docs` directory, with a rendered version [available online](#). This documentation is intended primarily for sysadmins running their own Synapse instance, as well as developers interacting externally with Synapse.

`docs/development` exists primarily to house documentation for Synapse developers.

`docs/admin_api` houses documentation regarding Synapse's Admin API, which is used mostly by sysadmins and external service developers.

Synapse's code style is documented [here](#). Please follow it, including the conventions for [configuration options and documentation](#).

We welcome improvements and additions to our documentation itself! When writing new pages, please [build docs](#) to a book to check that your contributions render correctly. The docs are written in [GitHub-Flavoured Markdown](#).

When changes are made to any Rust code then you must call either `poetry install` or `maturin develop` (if installed) to rebuild the Rust code. Using `maturin` is quicker than `poetry install`, so is recommended when making frequent changes to the Rust code.

8. Test, test, test!

While you're developing and before submitting a patch, you'll want to test your code.

Run the linters.

The linters look at your code and do two things:

- ensure that your code follows the coding style adopted by the project;
- catch a number of errors in your code.

The linter takes no time at all to run as soon as you've [downloaded the dependencies](#).

```
poetry run ./scripts-dev/lint.sh
```

Note that this script *will modify your files* to fix styling errors. Make sure that you have saved all your files.

If you wish to restrict the linters to only the files changed since the last commit (much faster!), you can instead run:

```
poetry run ./scripts-dev/lint.sh -d
```

Or if you know exactly which files you wish to lint, you can instead run:

```
poetry run ./scripts-dev/lint.sh path/to/file1.py path/to/file2.py  
path/to/folder
```

Run the unit tests (Twisted trial).

The unit tests run parts of Synapse, including your changes, to see if anything was broken. They are slower than the linters but will typically catch more errors.

```
poetry run trial tests
```

You can run unit tests in parallel by specifying `-jX` argument to `trial` where `X` is the number of parallel runners you want. To use 4 cpu cores, you would run them like:

```
poetry run trial -j4 tests
```

If you wish to only run *some* unit tests, you may specify another module instead of `tests` - or a test class or a method:

```
poetry run trial tests.rest.admin.test_room
tests.handlers.test_admin.ExfiltrateData.test_invite
```

If your tests fail, you may wish to look at the logs (the default log level is `ERROR`):

```
less _trial_temp/test.log
```

To increase the log level for the tests, set `SYNAPSE_TEST_LOG_LEVEL`:

```
SYNAPSE_TEST_LOG_LEVEL=DEBUG poetry run trial tests
```

By default, tests will use an in-memory SQLite database for test data. For additional help with debugging, one can use an on-disk SQLite database file instead, in order to review database state during and after running tests. This can be done by setting the `SYNAPSE_TEST_PERSIST_SQLITE_DB` environment variable. Doing so will cause the database state to be stored in a file named `test.db` under the trial process' working directory. Typically, this ends up being `_trial_temp/test.db`. For example:

```
SYNAPSE_TEST_PERSIST_SQLITE_DB=1 poetry run trial tests
```

The database file can then be inspected with:

```
sqlite3 _trial_temp/test.db
```

Note that the database file is cleared at the beginning of each test run. Thus it will always only contain the data generated by the *last run test*. Though generally when debugging, one is only running a single test anyway.

Running tests under PostgreSQL

Invoking `trial` as above will use an in-memory SQLite database. This is great for quick development and testing. However, we recommend using a PostgreSQL database in production (and indeed, we have some code paths specific to each database). This means that we need to run our unit tests against PostgreSQL too. Our CI does this automatically for pull requests and release candidates, but it's sometimes useful to reproduce this locally.

Using Docker

The easiest way to do so is to run Postgres via a docker container. In one terminal:

```
docker run --rm -e POSTGRES_PASSWORD=mysecretpassword -e POSTGRES_USER=postgres -e POSTGRES_DB=postgres -p 5432:5432 postgres:14
```

If you see an error like

```
docker: Error response from daemon: driver failed programming external connectivity on endpoint nice_ride (b57bbe2e251b70015518d00c9981e8cb8346b5c785250341a6c53e3c899875f1): Error starting userland proxy: listen tcp4 0.0.0.0:5432: bind: address already in use.
```

then something is already bound to port 5432. You're probably already running postgres locally.

Once you have a postgres server running, invoke `trial` in a second terminal:

```
SYNAPSE_POSTGRES=1 SYNAPSE_POSTGRES_HOST=127.0.0.1
SYNAPSE_POSTGRES_USER=postgres SYNAPSE_POSTGRES_PASSWORD=mysecretpassword poetry
run trial tests
```

Using an existing Postgres installation

If you have postgres already installed on your system, you can run `trial` with the following environment variables matching your configuration:

- `SYNAPSE_POSTGRES` to anything nonempty
- `SYNAPSE_POSTGRES_HOST` (optional if it's the default: UNIX socket)
- `SYNAPSE_POSTGRES_PORT` (optional if it's the default: 5432)
- `SYNAPSE_POSTGRES_USER` (optional if using a UNIX socket)
- `SYNAPSE_POSTGRES_PASSWORD` (optional if using a UNIX socket)

For example:

```
export SYNAPSE_POSTGRES=1
export SYNAPSE_POSTGRES_HOST=localhost
export SYNAPSE_POSTGRES_USER=postgres
export SYNAPSE_POSTGRES_PASSWORD=mydevenvpassword
trial
```

You don't need to specify the host, user, port or password if your Postgres server is set to authenticate you over the UNIX socket (i.e. if the `psql` command works without further arguments).

Your Postgres account needs to be able to create databases; see the postgres docs for `ALTER ROLE`.

Run the integration tests (Sytest).

The integration tests are a more comprehensive suite of tests. They run a full version of Synapse, including your changes, to check if anything was broken. They are slower than the unit tests but will typically catch more errors.

The following command will let you run the integration test with the most common configuration:

```
$ docker run --rm -it -v /path/where/you/have/cloned/the/repository\:/src:ro -v /path/to/where/you/want/logs\:/logs matrixdotorg/sytest-synapse:bullseye
```

(Note that the paths must be full paths! You could also write `$(realpath relative/path)` if needed.)

This configuration should generally cover your needs.

- To run with Postgres, supply the `-e POSTGRES=1 -e MULTI_POSTGRES=1` environment flags.
- To run with Synapse in worker mode, supply the `-e WORKERS=1 -e REDIS=1` environment flags (in addition to the Postgres flags).

For more details about other configurations, see the [Docker-specific documentation in the SyTest repo](#).

Run the integration tests (Complement).

[Complement](#) is a suite of black box tests that can be run on any homeserver implementation. It can also be thought of as end-to-end (e2e) tests.

It's often nice to develop on Synapse and write Complement tests at the same time. Here is how to run your local Synapse checkout against your local Complement checkout.

(checkout `complement` alongside your `synapse` checkout)

```
COMPLEMENT_DIR=../complement ./scripts-dev/complement.sh
```

To run a specific test file, you can pass the test name at the end of the command. The name passed comes from the naming structure in your Complement tests. If you're unsure of the name, you can do a full run and copy it from the test output:

```
COMPLEMENT_DIR=../complement ./scripts-dev/complement.sh -run
TestImportHistoricalMessages
```

To run a specific test, you can specify the whole name structure:

```
COMPLEMENT_DIR=../complement ./scripts-dev/complement.sh -run
TestImportHistoricalMessages/parallel/Historical_events_resolve_in_the_correct_order
```

The above will run a monolithic (single-process) Synapse with SQLite as the database. For other configurations, try:

- Passing `POSTGRES=1` as an environment variable to use the Postgres database instead.
- Passing `WORKERS=1` as an environment variable to use a workerised setup instead. This option implies the use of Postgres.
 - If setting `WORKERS=1`, optionally set `WORKER_TYPES=` to declare which worker types you wish to test. A simple comma-delimited string containing the worker types defined from the `WORKERS_CONFIG` template in [here](#). A safe example would be `WORKER_TYPES="federation_inbound, federation_sender, synchrotron"`. See the [worker documentation](#) for additional information on workers.
- Passing `ASYNCIO_REACTOR=1` as an environment variable to use the Twisted asyncio reactor instead of the default one.
- Passing `PODMAN=1` will use the `podman` container runtime, instead of docker.
- Passing `UNIX_SOCKETS=1` will utilise Unix socket functionality for Synapse, Redis, and Postgres (when applicable).

To increase the log level for the tests, set `SYNAPSE_TEST_LOG_LEVEL`, e.g:

```
SYNAPSE_TEST_LOG_LEVEL=DEBUG COMPLEMENT_DIR=../complement ./scripts-dev/complement.sh -run TestImportHistoricalMessages
```

Prettier formatting with `gotestfmt`

If you want to format the output of the tests the same way as it looks in CI, install [gotestfmt](#).

You can then use this incantation to format the tests appropriately:

```
COMPLEMENT_DIR=../complement ./scripts-dev/complement.sh -json | gotestfmt -hide
successful-tests
```

(Remove `-hide successful-tests` if you don't want to hide successful tests.)

Access database for homeserver after Complement test runs.

If you're curious what the database looks like after you run some tests, here are some steps to get you going in Synapse:

1. In your Complement test comment out `defer deployment.Destroy(t)` and replace with `defer time.Sleep(2 * time.Hour)` to keep the homeserver running after the tests complete

2. Start the Complement tests
3. Find the name of the container, `docker ps -f name=complement_` (this will filter for just the Complement related Docker containers)
4. Access the container replacing the name with what you found in the previous step:
`docker exec -it complement_1_hs_with_application_service.hs1_2 /bin/bash`
5. Install sqlite (database driver), `apt-get update && apt-get install -y sqlite3`
6. Then run `sqlite3` and open the database `.open /conf/homeserver.db` (this db path comes from the Synapse homeserver.yaml)

9. Submit your patch.

Once you're happy with your patch, it's time to prepare a Pull Request.

To prepare a Pull Request, please:

1. verify that [all the tests pass](#), including the coding style;
2. [sign off](#) your contribution;
3. `git push` your commit to your fork of Synapse;
4. on GitHub, [create the Pull Request](#);
5. add a [changelog entry](#) and push it to your Pull Request;
6. that's it for now, a non-draft pull request will automatically request review from the team;
7. if you need to update your PR, please avoid rebasing and just add new commits to your branch.

Changelog

All changes, even minor ones, need a corresponding changelog / newsfragment entry. These are managed by [Towncrier](#).

To create a changelog entry, make a new file in the `changelog.d` directory named in the format of `PRnumber.type`. The type can be one of the following:

- `feature`
- `bugfix`
- `docker` (for updates to the Docker image)
- `doc` (for updates to the documentation)
- `removal` (also used for deprecations)
- `misc` (for internal-only changes)

This file will become part of our [changelog](#) at the next release, so the content of the file should be a short description of your change in the same style as the rest of the changelog.

The file can contain Markdown formatting, and must end with a full stop (.) or an exclamation mark (!) for consistency.

Adding credits to the changelog is encouraged, we value your contributions and would like to have you shouted out in the release notes!

For example, a fix in PR #1234 would have its changelog entry in `changelog.d/1234.bugfix`, and contain content like:

The security levels of Florbs are now validated when received via the `/federation/florb` endpoint. Contributed by Jane Matrix.

If there are multiple pull requests involved in a single bugfix/feature/etc, then the content for each `changelog.d` file and file extension should be the same. Towncrier will merge the matching files together into a single changelog entry when we come to release.

How do I know what to call the changelog file before I create the PR?

Obviously, you don't know if you should call your newsfile `1234.bugfix` or `5678.bugfix` until you create the PR, which leads to a chicken-and-egg problem.

There are two options for solving this:

1. Open the PR without a changelog file, see what number you got, and *then* add the changelog file to your branch, or:
2. Look at the [list of all issues/PRs](#), add one to the highest number you see, and quickly open the PR before somebody else claims your number.

[This script](#) might be helpful if you find yourself doing this a lot.

Sorry, we know it's a bit fiddly, but it's *really* helpful for us when we come to put together a release!

Debian changelog

Changes which affect the debian packaging files (in `debian`) are an exception to the rule that all changes require a `changelog.d` file.

In this case, you will need to add an entry to the debian changelog for the next release. For this, run the following command:

```
dch
```

This will make up a new version number (if there isn't already an unreleased version in flight), and open an editor where you can add a new changelog entry. (Our release process will ensure that the version number and maintainer name is corrected for the release.)

If your change affects both the debian packaging *and* files outside the debian directory, you will need both a regular newsfragment *and* an entry in the debian changelog. (Though typically such changes should be submitted as two separate pull requests.)

Sign off

After you make a PR a comment from @CLAassistant will appear asking you to sign the [CLA](#). This will link a page to allow you to confirm that you have read and agreed to the CLA by signing in with GitHub.

Alternatively, you can sign off before opening a PR by going to <https://cla-assistant.io/element-hq/synapse>.

We accept contributions under a legally identifiable name, such as your name on government documentation or common-law names (names claimed by legitimate usage or repute). Unfortunately, we cannot accept anonymous contributions at this time.

10. Turn feedback into better code.

Once the Pull Request is opened, you will see a few things:

1. our automated CI (Continuous Integration) pipeline will run (again) the linters, the unit tests, the integration tests and more;
2. one or more of the developers will take a look at your Pull Request and offer feedback.

From this point, you should:

1. Look at the results of the CI pipeline.
 - o If there is any error, fix the error.
2. If a developer has requested changes, make these changes and let us know if it is ready for a developer to review again.
 - o A pull request is a conversation, if you disagree with the suggestions, please respond and discuss it.
3. Create a new commit with the changes.
 - o Please do NOT overwrite the history. New commits make the reviewer's life easier.
 - o Push this commits to your Pull Request.
4. Back to 1.
5. Once the pull request is ready for review again please re-request review from whichever developer did your initial review (or leave a comment in the pull request

that you believe all required changes have been done).

Once both the CI and the developers are happy, the patch will be merged into Synapse and released shortly!

11. Find a new issue.

By now, you know the drill!

Notes for maintainers on merging PRs etc

There are some notes for those with commit access to the project on how we manage git [here](#).

Conclusion

That's it! Matrix is a very open and collaborative project as you might expect given our obsession with open communication. If we're going to successfully matrix together all the fragmented communication technologies out there we are reliant on contributions and collaboration from the community to do so. So please get involved - and we hope you have as much fun hacking on Matrix as we do!

Code Style

Formatting tools

The Synapse codebase uses a number of code formatting tools in order to quickly and automatically check for formatting (and sometimes logical) errors in code.

The necessary tools are:

- [ruff](#), which can spot common errors and enforce a consistent style; and
- [mypy](#), a type checker.

See [the contributing guide](#) for instructions on how to install the above tools and run the linters.

It's worth noting that modern IDEs and text editors can run these tools automatically on save. It may be worth looking into whether this functionality is supported in your editor for a more convenient development workflow. It is not, however, recommended to run [mypy](#) on save as it takes a while and can be very resource intensive.

General rules

- **Naming:**
 - Use `CamelCase` for class and type names
 - Use underscores for `function_names` and `variable_names`.
- **Docstrings:** should follow the [google code style](#). See the [examples](#) in the sphinx documentation.
- **Imports:**
 - Imports should be sorted by `isort` as described above.
 - Prefer to import classes and functions rather than packages or modules.

Example:

```
from synapse.types import UserID
...
user_id = UserID(local, server)
```

is preferred over:

```
from synapse import types
...
user_id = types.UserID(local, server)
```

(or any other variant).

This goes against the advice in the Google style guide, but it means that errors in the name are caught early (at import time).

- Avoid wildcard imports (`from synapse.types import *`) and relative imports (`(from .types import UserID)`).

Configuration code and documentation format

When adding a configuration option to the code, if several settings are grouped into a single dict, ensure that your code correctly handles the top-level option being set to `None` (as it will be if no sub-options are enabled).

The [configuration manual](#) acts as a reference to Synapse's configuration options for server administrators. Remember that many readers will be unfamiliar with YAML and server administration in general, so it is important that when you add a configuration option the documentation be as easy to understand as possible, which includes following a consistent format.

Some guidelines follow:

- Each option should be listed in the config manual with the following format:
 - The name of the option, prefixed by `###`.
 - A comment which describes the default behaviour (i.e. what happens if the setting is omitted), as well as what the effect will be if the setting is changed.
 - An example setting, using backticks to define the code block

For boolean (on/off) options, convention is that this example should be the *opposite* to the default. For other options, the example should give some non-default value which is likely to be useful to the reader.

- There should be a horizontal rule between each option, which can be achieved by adding `---` before and after the option.
- `true` and `false` are spelt thus (as opposed to `True`, etc.)

Example:

modules

Use the `module` sub-option to add a module under `modules` to extend functionality. The `module` setting then has a sub-option, `config`, which can be used to define some configuration for the `module`.

Defaults to none.

Example configuration:

```
modules:
- module: my_super_module.MySuperClass
  config:
    do_thing: true
- module: my_other_super_module.SomeClass
  config: {}
```

Note that the sample configuration is generated from the synapse code and is maintained by a script, `scripts-dev/generate_sample_config.sh`. Making sure that the output from this script matches the desired format is left as an exercise for the reader!

Some notes on how we do reviews

The Synapse team works off a shared review queue -- any new pull requests for Synapse (or related projects) has a review requested from the entire team. Team members should process this queue using the following rules:

- Any high urgency pull requests (e.g. fixes for broken continuous integration or fixes for release blockers);
- Follow-up reviews for pull requests which have previously received reviews;
- Any remaining pull requests.

For the latter two categories above, older pull requests should be prioritised.

It is explicit that there is no priority given to pull requests from the team (vs from the community). If a pull request requires a quick turn around, please explicitly communicate this via [#synapse-dev:matrix.org](#) or as a comment on the pull request.

Once an initial review has been completed and the author has made additional changes, follow-up reviews should go back to the same reviewer. This helps build a shared context and conversation between author and reviewer.

As a team we aim to keep the number of inflight pull requests to a minimum to ensure that ongoing work is finished before starting new work.

Performing a review

To communicate to the rest of the team the status of each pull request, team members should do the following:

- Assign themselves to the pull request (they should be left assigned to the pull request until it is merged, closed, or are no longer the reviewer);
- Review the pull request by leaving comments, questions, and suggestions;
- Mark the pull request appropriately (as needing changes or accepted).

If you are unsure about a particular part of the pull request (or are not confident in your understanding of part of the code) then ask questions or request review from the team again. When requesting review from the team be sure to leave a comment with the rationale on why you're putting it back in the queue.

Synapse Release Cycle

Releases of Synapse follow a two week release cycle with new releases usually occurring on Tuesdays:

- Day 0: Synapse $N - 1$ is released.
- Day 7: Synapse N release candidate 1 is released.
- Days 7 - 13: Synapse N release candidates 2+ are released, if bugs are found.
- Day 14: Synapse N is released.

Note that this schedule might be modified depending on the availability of the Synapse team, e.g. releases may be skipped to avoid holidays.

Release announcements can be found in the [release category of the Matrix blog](#).

Bugfix releases

If a bug is found after release that is deemed severe enough (by a combination of the impacted users and the impact on those users) then a bugfix release may be issued. This may be at any point in the release cycle.

Security releases

Security will sometimes be backported to the previous version and released immediately before the next release candidate. An example of this might be:

- Day 0: Synapse $N - 1$ is released.
- Day 7: Synapse $(N - 1).1$ is released as Synapse $N - 1$ + the security fix.
- Day 7: Synapse N release candidate 1 is released (including the security fix).

Depending on the impact and complexity of security fixes, multiple fixes might be held to be released together.

In some cases, a pre-disclosure of a security release will be issued as a notice to Synapse operators that there is an upcoming security release. These can be found in the [security category of the Matrix blog](#).

Some notes on how we use git

On keeping the commit history clean

In an ideal world, our git commit history would be a linear progression of commits each of which contains a single change building on what came before. Here, by way of an arbitrary example, is the top of `git log --graph b2dba0607`:

```
* commit b2dba0607944107990883618672d639f0614c492
| Author: Richard van der Hoff <1389908+richvdh@users.noreply.github.com>
| Date:   Fri May 1 09:25:16 2020 +0100
|
|   Workaround for assertion errors from db_query_to_update_function (#7378)
|
|   Hopefully this is no worse than what we have on master...
|
* commit 627b0f5f2753e6910adb7a877541d50f5936b8a5
| Author: Patrick Cloke <clokep@users.noreply.github.com>
| Date:   Thu Apr 30 13:47:49 2020 -0400
|
|   Persist user interactive authentication sessions (#7302)
|
|   By persisting the user interactive authentication sessions to the database, this fixes
|   situations where a user hits different works throughout their auth session and also
|   allows sessions to persist through restarts of Synapse.
|
* commit 9d8ecc9e6c48b9dfc0b41326794b8e10fc6ad062
| Author: Andrew Morgan <1342360+anoadragon453@users.noreply.github.com>
| Date:   Thu Apr 30 11:38:07 2020 +0100
|
|   Apply federation check for /publicRooms with filter list (#7367)
|
* commit 37f6823f5b91f27b9dd8de8fc0e52d5ea889647c
| Author: Erik Johnston <erik@matrix.org>
| Date:   Wed Apr 29 16:23:08 2020 +0100
|
|   Add instance name to RDATA/POSITION commands (#7364)
|
|   This is primarily for allowing us to send those commands from workers, but for now
|   simply allows us to ignore echoed RDATA/POSITION commands that we sent (we get echoes
|   of sent commands when using redis). Currently we log a WARNING on the master process
|   every time we receive an echoed RDATA.
|
* commit 3eab76ad43e1de4074550c468d11318d497ff632
| Author: Erik Johnston <erik@matrix.org>
| Date:   Wed Apr 29 14:10:59 2020 +0100
|
|   Don't relay REMOTE_SERVER_UP cmds to same conn. (#7352)
|
|   For direct TCP connections we need the master to relay REMOTE_SERVER_UP
|   commands to the other connections so that all instances get notified
|   about it. The old implementation just relayed to all connections,
|   assuming that sending back to the original sender of the command was
|   safe. This is not true for redis, where commands sent get echoed back to
|   the sender, which was causing master to effectively infinite loop
|   sending and then re-receiving REMOTE_SERVER_UP commands that it sent.
|
|   The fix is to ensure that we only relay to *other* connections and not
|   to the connection we received the notification from.
|
|   Fixes #7334.
```

Note how the commit comment explains clearly what is changing and why. Also note the *absence* of merge commits, as well as the absence of commits called things like (to pick a few culprits): “pep8”, “fix broken test”, “oops”, “typo”, or “Who's the president?”.

There are a number of reasons why keeping a clean commit history is a good thing:

- From time to time, after a change lands, it turns out to be necessary to revert it, or to backport it to a release branch. Those operations are *much* easier when the change is contained in a single commit.

- Similarly, it's much easier to answer questions like "is the fix for `/publicRooms` on the release branch?" if that change consists of a single commit.
- Likewise: "what has changed on this branch in the last week?" is much clearer without merges and "pep8" commits everywhere.
- Sometimes we need to figure out where a bug got introduced, or some behaviour changed. One way of doing that is with `git bisect`: pick an arbitrary commit between the known good point and the known bad point, and see how the code behaves. However, that strategy fails if the commit you chose is the middle of someone's epic branch in which they broke the world before putting it back together again.

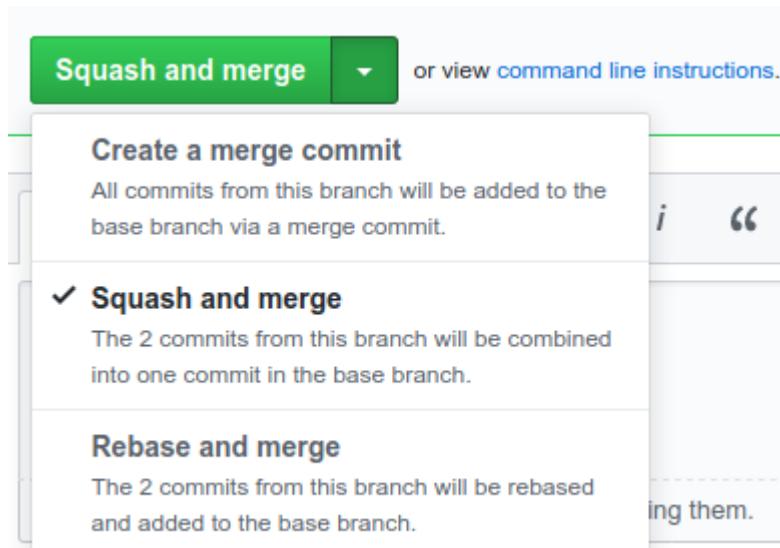
One counterargument is that it is sometimes useful to see how a PR evolved as it went through review cycles. This is true, but that information is always available via the GitHub UI (or via the little-known `refs/pull` namespace).

Of course, in reality, things are more complicated than that. We have release branches as well as `develop` and `master`, and we deliberately merge changes between them. Bugs often slip through and have to be fixed later. That's all fine: this not a cast-iron rule which must be obeyed, but an ideal to aim towards.

Merges, squashes, rebases: wtf?

Ok, so that's what we'd like to achieve. How do we achieve it?

The TL;DR is: when you come to merge a pull request, you *probably* want to "squash and merge":



(This applies whether you are merging your own PR, or that of another contributor.)

“Squash and merge”¹ takes all of the changes in the PR, and bundles them into a single commit. GitHub gives you the opportunity to edit the commit message before you confirm, and normally you should do so, because the default will be useless (again: `* woops typo` is not a useful thing to keep in the historical record).

The main problem with this approach comes when you have a series of pull requests which build on top of one another: as soon as you squash-merge the first PR, you'll end up with a stack of conflicts to resolve in all of the others. In general, it's best to avoid this situation in the first place by trying not to have multiple related PRs in flight at the same time. Still, sometimes that's not possible and doing a regular merge is the lesser evil.

Another occasion in which a regular merge makes more sense is a PR where you've deliberately created a series of commits each of which makes sense in its own right. For example: a PR which gradually propagates a refactoring operation through the codebase, or a PR which is the culmination of several other PRs. In this case the ability to figure out when a particular change/bug was introduced could be very useful.

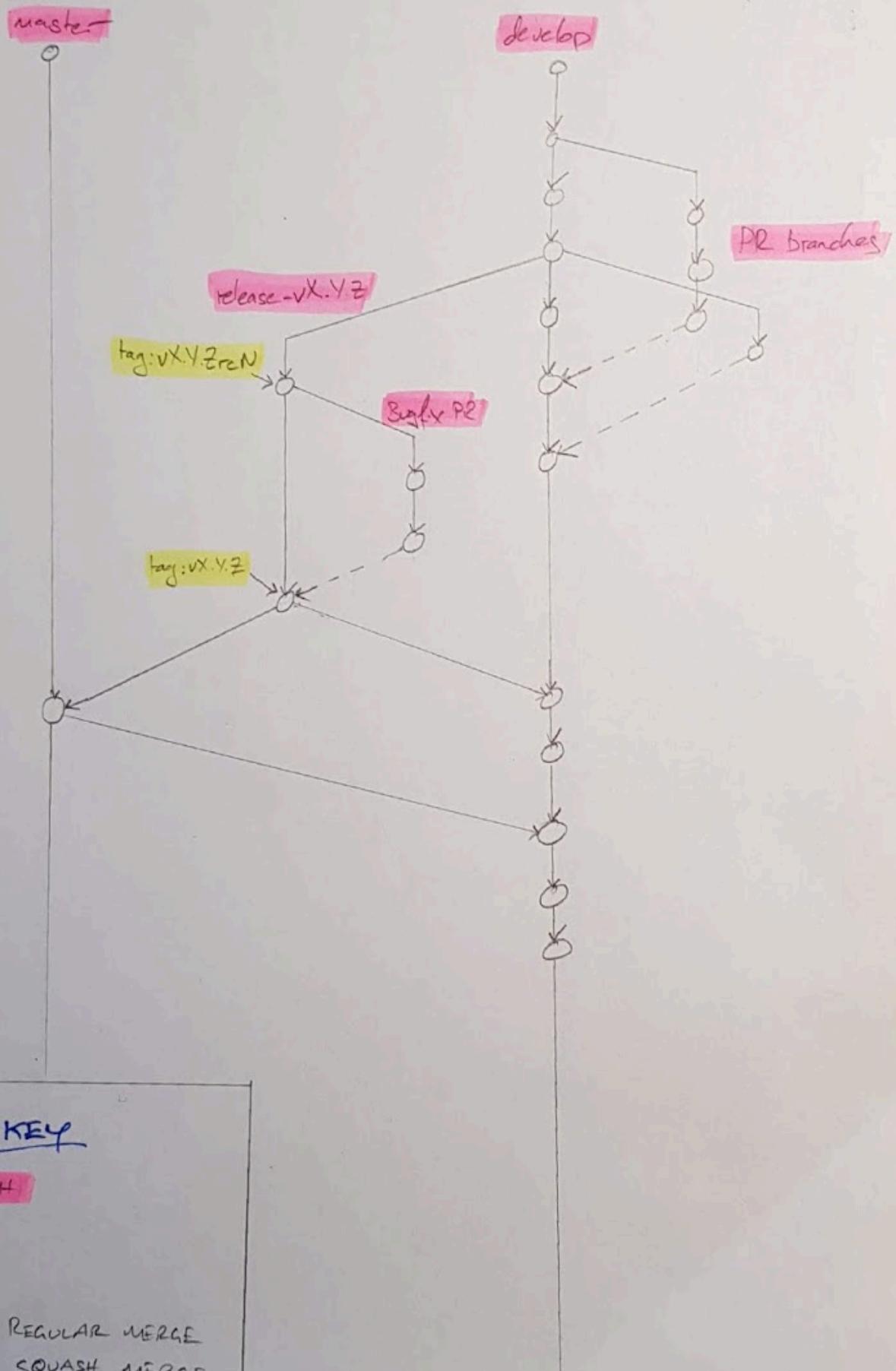
Ultimately: **this is not a hard-and-fast-rule**. If in doubt, ask yourself “do each of the commits I am about to merge make sense in their own right”, but remember that we're just doing our best to balance “keeping the commit history clean” with other factors.

Git branching model

A lot of words have been written in the past about git branching models (no really, a lot). I tend to think the whole thing is overblown. Fundamentally, it's not that complicated. Here's how we do it.

Let's start with a picture:

More Stable $\dots \dots \dots \dots \dots \dots \dots \rightarrow$ Less Stable



It looks complicated, but it's really not. There's one basic rule: *anyone* is free to merge from *any* more-stable branch to *any* less-stable branch at *any* time². (The principle behind this is that if a change is good enough for the more-stable branch, then it's also good enough to put in a less-stable branch.)

Meanwhile, merging (or squashing, as per the above) from a less-stable to a more-stable branch is a deliberate action in which you want to publish a change or a set of changes to (some subset of) the world: for example, this happens when a PR is landed, or as part of our release process.

So, what counts as a more- or less-stable branch? A little reflection will show that our active branches are ordered thus, from more-stable to less-stable:

- `master` (tracks our last release).
- `release-vX.Y` (the branch where we prepare the next release)³.
- PR branches which are targeting the release.
- `develop` (our "mainline" branch containing our bleeding-edge).
- regular PR branches.

The corollary is: if you have a bugfix that needs to land in both `release-vX.Y` and `develop`, then you should base your PR on `release-vX.Y`, get it merged there, and then merge from `release-vX.Y` to `develop`. (If a fix lands in `develop` and we later need it in a release-branch, we can of course cherry-pick it, but landing it in the release branch first helps reduce the chance of annoying conflicts.)

[1]: "Squash and merge" is GitHub's term for this operation. Given that there is no merge involved, I'm not convinced it's the most intuitive name. [▲]

[2]: Well, anyone with commit access. [▲]

[3]: Very, very occasionally (I think this has happened once in the history of Synapse), we've had two releases in flight at once. Obviously, `release-v1.2` is more-stable than `release-v1.3`. [▲]

Synapse demo setup

DO NOT USE THESE DEMO SERVERS IN PRODUCTION

Requires you to have a [Synapse development environment setup](#).

The demo setup allows running three federation Synapse servers, with server names `localhost:8480`, `localhost:8481`, and `localhost:8482`.

You can access them via any Matrix client over HTTP at `localhost:8080`, `localhost:8081`, and `localhost:8082` or over HTTPS at `localhost:8480`, `localhost:8481`, and `localhost:8482`.

To enable the servers to communicate, self-signed SSL certificates are generated and the servers are configured in a highly insecure way, including:

- Not checking certificates over federation.
- Not verifying keys.

The servers are configured to store their data under `demo/8080`, `demo/8081`, and `demo/8082`. This includes configuration, logs, SQLite databases, and media.

Note that when joining a public room on a different homeserver via "#foo:bar.net", then you are (in the current implementation) joining a room with room_id "foo". This means that it won't work if your homeserver already has a room with that name.

Using the demo scripts

There's three main scripts with straightforward purposes:

- `start.sh` will start the Synapse servers, generating any missing configuration.
 - This accepts a single parameter `--no-rate-limit` to "disable" rate limits (they actually still exist, but are very high).
- `stop.sh` will stop the Synapse servers.
- `clean.sh` will delete the configuration, databases, log files, etc.

To start a completely new set of servers, run:

```
./demo/stop.sh; ./demo/clean.sh && ./demo/start.sh
```

OpenTracing

Background

OpenTracing is a semi-standard being adopted by a number of distributed tracing platforms. It is a common api for facilitating vendor-agnostic tracing instrumentation. That is, we can use the OpenTracing api and select one of a number of tracer implementations to do the heavy lifting in the background. Our current selected implementation is Jaeger.

OpenTracing is a tool which gives an insight into the causal relationship of work done in and between servers. The servers each track events and report them to a centralised server - in Synapse's case: Jaeger. The basic unit used to represent events is the span. The span roughly represents a single piece of work that was done and the time at which it occurred. A span can have child spans, meaning that the work of the child had to be completed for the parent span to complete, or it can have follow-on spans which represent work that is undertaken as a result of the parent but is not depended on by the parent to in order to finish.

Since this is undertaken in a distributed environment a request to another server, such as an RPC or a simple GET, can be considered a span (a unit of work) for the local server. This causal link is what OpenTracing aims to capture and visualise. In order to do this metadata about the local server's span, i.e the 'span context', needs to be included with the request to the remote.

It is up to the remote server to decide what it does with the spans it creates. This is called the sampling policy and it can be configured through Jaeger's settings.

For OpenTracing concepts see <https://opentracing.io/docs/overview/what-is-tracing/>.

For more information about Jaeger's implementation see <https://www.jaegertracing.io/docs/>

Setting up OpenTracing

To receive OpenTracing spans, start up a Jaeger server. This can be done using docker like so:

```
docker run -d --name jaeger \
-p 6831:6831/udp \
-p 6832:6832/udp \
-p 5778:5778 \
-p 16686:16686 \
-p 14268:14268 \
jaegertracing/all-in-one:1
```

By default, Synapse will publish traces to Jaeger on localhost. If Jaeger is hosted elsewhere, point Synapse to the correct host by setting

`opentracing.jaeger_config.local_agent.reporting_host` in the [Synapse configuration](#) or by setting the `JAEGER_AGENT_HOST` environment variable to the desired address.

Latest documentation is probably at <https://www.jaegertracing.io/docs/latest/getting-started>.

Enable OpenTracing in Synapse

OpenTracing is not enabled by default. It must be enabled in the homeserver config by adding the `opentracing` option to your config file. You can find documentation about how to do this in the [config manual](#) under the header 'Opentracing'. See below for an example OpenTracing configuration:

```
opentracing:
  enabled: true
  homeserver_whitelist:
    - "mytrustedhomeserver.org"
    - "*.myotherhomeservers.com"
```

Homeserver whitelisting

The homeserver whitelist is configured using regular expressions. A list of regular expressions can be given and their union will be compared when propagating any spans contexts to another homeserver.

Though it's mostly safe to send and receive span contexts to and from untrusted users since span contexts are usually opaque ids it can lead to two problems, namely:

- If the span context is marked as sampled by the sending homeserver the receiver will sample it. Therefore two homeservers with wildly different sampling policies could incur higher sampling counts than intended.
- Sending servers can attach arbitrary data to spans, known as 'baggage'. For safety this has been disabled in Synapse but that doesn't prevent another server sending you baggage which will be logged to OpenTracing's logs.

Configuring Jaeger

Sampling strategies can be set as in this document:

<https://www.jaegertracing.io/docs/latest/sampling/>.

Synapse database schema files

Synapse's database schema is stored in the `synapse.storage.schema` module.

Logical databases

Synapse supports splitting its datastore across multiple physical databases (which can be useful for large installations), and the schema files are therefore split according to the logical database they apply to.

At the time of writing, the following "logical" databases are supported:

- `state` - used to store Matrix room state (more specifically, `state_groups`, their relationships and contents).
- `main` - stores everything else.

Additionally, the `common` directory contains schema files for tables which must be present on *all* physical databases.

Synapse schema versions

Synapse manages its database schema via "schema versions". These are mainly used to help avoid confusion if the Synapse codebase is rolled back after the database is updated. They work as follows:

- The Synapse codebase defines a constant `synapse.storage.schema.SCHEMA_VERSION` which represents the expectations made about the database by that version. For example, as of Synapse v1.36, this is `59`.
- The database stores a "compatibility version" in `schema_compat_version.compat_version` which defines the `SCHEMA_VERSION` of the oldest version of Synapse which will work with the database. On startup, if `compat_version` is found to be newer than `SCHEMA_VERSION`, Synapse will refuse to start.

Synapse automatically updates this field from `synapse.storage.schema.SCHEMA_COMPAT_VERSION`.

- Whenever a backwards-incompatible change is made to the database format (normally via a `delta` file), `synapse.storage.schema.SCHEMA_COMPAT_VERSION` is also updated so that administrators can not accidentally roll back to a too-old version of Synapse.

Generally, the goal is to maintain compatibility with at least one or two previous releases of Synapse, so any substantial change tends to require multiple releases and a bit of forward-planning to get right.

As a worked example: we want to remove the `room_stats_historical` table. Here is how it might pan out.

1. Replace any code that *reads* from `room_stats_historical` with alternative implementations, but keep writing to it in case of rollback to an earlier version. Also, increase `synapse.storage.schema.SCHEMA_VERSION`. In this instance, there is no existing code which reads from `room_stats_historical`, so our starting point is:

v1.36.0: `SCHEMA_VERSION=59`, `SCHEMA_COMPAT_VERSION=59`

2. Next (say in Synapse v1.37.0): remove the code that *writes* to `room_stats_historical`, but don't yet remove the table in case of rollback to v1.36.0. Again, we increase `synapse.storage.schema.SCHEMA_VERSION`, but because we have not broken compatibility with v1.36, we do not yet update `SCHEMA_COMPAT_VERSION`. We now have:

v1.37.0: `SCHEMA_VERSION=60`, `SCHEMA_COMPAT_VERSION=59`.

3. Later (say in Synapse v1.38.0): we can remove the table altogether. This will break compatibility with v1.36.0, so we must update `SCHEMA_COMPAT_VERSION` accordingly. There is no need to update `synapse.storage.schema.SCHEMA_VERSION`, since there is no change to the Synapse codebase here. So we end up with:

v1.38.0: `SCHEMA_VERSION=60`, `SCHEMA_COMPAT_VERSION=60`.

If in doubt about whether to update `SCHEMA_VERSION` or not, it is generally best to lean towards doing so.

Full schema dumps

In the `full_schemas` directories, only the most recently-numbered snapshot is used (54 at the time of writing). Older snapshots (eg, 16) are present for historical reference only.

Building full schema dumps

If you want to recreate these schemas, they need to be made from a database that has had all background updates run.

To do so, use `scripts-dev/make_full_schema.sh`. This will produce new `full.sql.postgres` and `full.sql.sqlite` files.

Ensure postgres is installed, then run:

```
./scripts-dev/make_full_schema.sh -p postgres_username -o output_dir/
```

NB at the time of writing, this script predates the split into separate `state`/`main` databases so will require updates to handle that correctly.

Delta files

Delta files define the steps required to upgrade the database from an earlier version. They can be written as either a file containing a series of SQL statements, or a Python module.

Synapse remembers which delta files it has applied to a database (they are stored in the `applied_schema_deltas` table) and will not re-apply them (even if a given file is subsequently updated).

Delta files should be placed in a directory named

`synapse/storage/schema/<database>/delta/<version>/`. They are applied in alphanumeric order, so by convention the first two characters of the filename should be an integer such as `01`, to put the file in the right order.

SQL delta files

These should be named `*.sql`, or — for changes which should only be applied for a given database engine — `*.sql.postgres` or `*.sql.sqlite`. For example, a delta which adds a new column to the `foo` table might be called `01add_bar_to_foo.sql`.

Note that our SQL parser is a bit simple - it understands comments (`--` and `/*...*/`), but complex statements which require a `;` in the middle of them (such as `CREATE TRIGGER`) are beyond it and you'll have to use a Python delta file.

Python delta files

For more flexibility, a delta file can take the form of a python module. These should be named `*.py`. Note that database-engine-specific modules are not supported here – instead you can write `if isinstance(database_engine, PostgresEngine)` or similar.

A Python delta module should define either or both of the following functions:

```

import synapse.config.homeserver
import synapse.storage.engines
import synapse.storage.types

def run_create(
    cur: synapse.storage.types.Cursor,
    database_engine: synapse.storage.engines.BaseDatabaseEngine,
) -> None:
    """Called whenever an existing or new database is to be upgraded"""
    ...

def run_upgrade(
    cur: synapse.storage.types.Cursor,
    database_engine: synapse.storage.engines.BaseDatabaseEngine,
    config: synapse.config.homeserver.HomeServerConfig,
) -> None:
    """Called whenever an existing database is to be upgraded."""
    ...

```

Background updates

It is sometimes appropriate to perform database migrations as part of a background process (instead of blocking Synapse until the migration is done). In particular, this is useful for migrating data when adding new columns or tables.

Pending background updates stored in the `background_updates` table and are denoted by a unique name, the current status (stored in JSON), and some dependency information:

- Whether the update requires a previous update to be complete.
- A rough ordering for which to complete updates.

A new background updates needs to be added to the `background_updates` table:

```

INSERT INTO background_updates (ordering, update_name, depends_on,
progress_json) VALUES
(7706, 'my_background_update', 'a_previous_background_update' '{}');

```

And then needs an associated handler in the appropriate datastore:

```

self.db_pool.updates.register_background_update_handler(
    "my_background_update",
    update_handler=self._my_background_update,
)

```

There are a few types of updates that can be performed, see the `BackgroundUpdater`:

- `register_background_update_handler`: A generic handler for custom SQL
- `register_background_index_update`: Create an index in the background

- `register_background_validate_constraint`: Validate a constraint in the background (PostgreSQL-only)
- `register_background_validate_constraint_and_delete_rows`: Similar to `register_background_validate_constraint`, but deletes rows which don't fit the constraint.

For `register_background_update_handler`, the generic handler must track progress and then finalize the background update:

```
async def _my_background_update(self, progress: JsonDict, batch_size: int) -> int:
    def _do_something(txn: LoggingTransaction) -> int:
        ...
        self.db_pool.updates._background_update_progress_txn(
            txn, "my_background_update", {"last_processed": last_processed})
    )
    return last_processed - prev_last_processed

    num_processed = await self.db_pool.runInteraction("_do_something",
    _do_something)
    await self.db_pool.updates._end_background_update("my_background_update")

    return num_processed
```

Synapse will attempt to rate-limit how often background updates are run via the given batch-size and the returned number of processed entries (and how long the function took to run). See [background update controller callbacks](#).

Boolean columns

Boolean columns require special treatment, since SQLite treats booleans the same as integers.

Any new boolean column must be added to the `BOOLEAN_COLUMNS` list in `synapse/_scripts/synapse_port_db.py`. This tells the port script to cast the integer value from SQLite to a boolean before writing the value to the postgres database.

event_id global uniqueness

`event_id`'s can be considered globally unique although there has been a lot of debate on this topic in places like [MSC2779](#) and [MSC2848](#) which has no resolution yet (as of 2022-09-01). There are several places in Synapse and even in the Matrix APIs like `GET /_matrix/federation/v1/event/{eventId}` where we assume that event IDs are globally unique.

When scoping `event_id` in a database schema, it is often nice to accompany it with `room_id` (`PRIMARY KEY (room_id, event_id)`) and a `FOREIGN KEY(room_id) REFERENCES rooms(room_id)`) which makes flexible lookups easy. For example it makes it very easy to find and clean up everything in a room when it needs to be purged (no need to use `sub-select` query or join from the `events` table).

A note on collisions: In room versions `1` and `2` it's possible to end up with two events with the same `event_id` (in the same or different rooms). After room version `3`, that can only happen with a hash collision, which we basically hope will never happen (SHA256 has a massive big key space).

Worked examples of gradual migrations

Some migrations need to be performed gradually. A prime example of this is anything which would need to do a large table scan — including adding columns, indices or `NOT NULL` constraints to non-empty tables — such a migration should be done as a background update where possible, at least on Postgres. We can afford to be more relaxed about SQLite databases since they are usually used on smaller deployments and SQLite does not support the same concurrent DDL operations as Postgres.

We also typically insist on having at least one Synapse version's worth of backwards compatibility, so that administrators can roll back Synapse if an upgrade did not go smoothly.

This sometimes results in having to plan a migration across multiple versions of Synapse.

This section includes an example and may include more in the future.

Transforming a column into another one, with `NOT NULL` constraints

This example illustrates how you would introduce a new column, write data into it based on data from an old column and then drop the old column.

We are aiming for semantic equivalence to:

```
ALTER TABLE mytable ADD COLUMN new_column INTEGER;
UPDATE mytable SET new_column = old_column * 100;
ALTER TABLE mytable ALTER COLUMN new_column ADD CONSTRAINT NOT NULL;
ALTER TABLE mytable DROP COLUMN old_column;
```

Synapse version N

```
SCHEMA_VERSION = S
SCHEMA_COMPAT_VERSION = ... # unimportant at this stage
```

Invariants:

1. `old_column` is read by Synapse and written to by Synapse.

Synapse version N + 1

```
SCHEMA_VERSION = S + 1
SCHEMA_COMPAT_VERSION = ... # unimportant at this stage
```

Changes:

1. `ALTER TABLE mytable ADD COLUMN new_column INTEGER;`

Invariants:

1. `old_column` is read by Synapse and written to by Synapse.
2. `new_column` is written to by Synapse.

Notes:

1. `new_column` can't have a `NOT NULL NOT VALID` constraint yet, because the previous Synapse version did not write to the new column (since we haven't bumped the `SCHEMA_COMPAT_VERSION` yet, we still need to be compatible with the previous version).

Synapse version N + 2

```
SCHEMA_VERSION = S + 2
SCHEMA_COMPAT_VERSION = S + 1 # this signals that we can't roll back to a time
before new_column existed
```

Changes:

1. On Postgres, add a `NOT VALID` constraint to ensure new rows are compliant. *SQLite does not have such a construct, but it would be unnecessary anyway since there is no way to concurrently perform this migration on SQLite.*

```
ALTER TABLE mytable ADD CONSTRAINT CHECK new_column_not_null (new_column IS
NOT NULL) NOT VALID;
```

2. Start a background update to perform migration: it should gradually run e.g.

```
UPDATE mytable SET new_column = old_column * 100 WHERE 0 < mytable_id AND
mytable_id <= 5;
```

This background update is technically pointless on SQLite, but you must schedule it anyway so that the `portdb` script to migrate to Postgres still works.

- Upon completion of the background update, you should run `VALIDATE CONSTRAINT` on Postgres to turn the `NOT VALID` constraint into a valid one.

```
ALTER TABLE mytable VALIDATE CONSTRAINT new_column_not_null;
```

This will take some time but does **NOT** hold an exclusive lock over the table.

Invariants:

- `old_column` is read by Synapse and written to by Synapse.
- `new_column` is written to by Synapse and new rows always have a non-`NULL` value in this field.

Notes:

- If you wish, you can convert the `CHECK (new_column IS NOT NULL)` to a `NOT NULL` constraint free of charge in Postgres by adding the `NOT NULL` constraint and then dropping the `CHECK` constraint, because Postgres can statically verify that the `NOT NULL` constraint is implied by the `CHECK` constraint without performing a table scan.
- It might be tempting to make version `N + 2` redundant by moving the background update to `N + 1` and delaying adding the `NOT NULL` constraint to `N + 3`, but that would mean the constraint would always be validated in the foreground in `N + 3`. Whereas if the `N + 2` step is kept, the migration in `N + 3` would be fast in the happy case.

Synapse version `N + 3`

```
SCHEMA_VERSION = S + 3
SCHEMA_COMPAT_VERSION = S + 1 # we can't roll back to a time before new_column
existed
```

Changes:

- (Postgres) Update the table to populate values of `new_column` in case the background update had not completed. Additionally, `VALIDATE CONSTRAINT` to make the check fully valid.

```
-- you ideally want an index on `new_column` or e.g. `(new_column) WHERE
new_column IS NULL` first, or perhaps you can find a way to skip this if
the `NOT NULL` constraint has already been validated.

UPDATE mytable SET new_column = old_column * 100 WHERE new_column IS NULL;

-- this is a no-op if it already ran as part of the background update
ALTER TABLE mytable VALIDATE CONSTRAINT new_column_not_null;
```

2. (SQLite) Recreate the table by precisely following [the 12-step procedure for SQLite table schema changes](#). During this table rewrite, you should recreate `new_column` as `NOT NULL` and populate any outstanding `NULL` values at the same time. Unfortunately, you can't drop `old_column` yet because it must be present for compatibility with the Postgres schema, as needed by `portdb`. (Otherwise you could do this all in one go with SQLite!)

Invariants:

1. `old_column` is written to by Synapse (but no longer read by Synapse!).
2. `new_column` is read by Synapse and written to by Synapse. Moreover, all rows have a non-`NULL` value in this field, as guaranteed by a schema constraint.

Notes:

1. We can't drop `old_column` yet, or even stop writing to it, because that would break a rollback to the previous version of Synapse.
2. Application code can now rely on `new_column` being populated. The remaining steps are only motivated by the wish to clean-up old columns.

Synapse version `N + 4`

```
SCHEMA_VERSION = S + 4
SCHEMA_COMPAT_VERSION = S + 3 # we can't roll back to a time before new_column
was entirely non-NULL
```

Invariants:

1. `old_column` exists but is not written to or read from by Synapse.
2. `new_column` is read by Synapse and written to by Synapse. Moreover, all rows have a non-`NULL` value in this field, as guaranteed by a schema constraint.

Notes:

1. We can't drop `old_column` yet because that would break a rollback to the previous version of Synapse.
- TODO:** It may be possible to relax this and drop the column straight away as long as the previous version of Synapse detected a rollback occurred and stopped attempting

to write to the column. This could possibly be done by checking whether the database's schema compatibility version was `S + 3`.

Synapse version `N + 5`

```
SCHEMA_VERSION = S + 5
SCHEMA_COMPAT_VERSION = S + 4 # we can't roll back to a time before old_column
was no longer being touched
```

Changes:

1. `ALTER TABLE mytable DROP COLUMN old_column;`

Implementing experimental features in Synapse

It can be desirable to implement "experimental" features which are disabled by default and must be explicitly enabled via the Synapse configuration. This is applicable for features which:

- Are unstable in the Matrix spec (e.g. those defined by an MSC that has not yet been merged).
- Developers are not confident in their use by general Synapse administrators/users (e.g. a feature is incomplete, buggy, performs poorly, or needs further testing).

Note that this only really applies to features which are expected to be desirable to a broad audience. The [module infrastructure](#) should instead be investigated for non-standard features.

Guarding experimental features behind configuration flags should help with some of the following scenarios:

- Ensure that clients do not assume that unstable features exist (failing gracefully if they do not).
- Unstable features do not become de-facto standards and can be removed aggressively (since only those who have opted-in will be affected).
- Ease finding the implementation of unstable features in Synapse (for future removal or stabilization).
- Ease testing a feature (or removal of feature) due to enabling/disabling without code changes. It also becomes possible to ask for wider testing, if desired.

Experimental configuration flags should be disabled by default (requiring Synapse administrators to explicitly opt-in), although there are situations where it makes sense (from a product point-of-view) to enable features by default. This is expected and not an issue.

It is not a requirement for experimental features to be behind a configuration flag, but one should be used if unsure.

New experimental configuration flags should be added under the `experimental` configuration key (see the `synapse.config.experimental` file) and either explain (briefly) what is being enabled, or include the MSC number.

Managing dependencies with Poetry

This is a quick cheat sheet for developers on how to use `poetry`.

Installing

See the [contributing guide](#).

Developers should use Poetry 1.3.2 or higher. If you encounter problems related to poetry, please [double-check your poetry version](#).

Background

Synapse uses a variety of third-party Python packages to function as a homeserver. Some of these are direct dependencies, listed in `pyproject.toml` under the `[tool.poetry.dependencies]` section. The rest are transitive dependencies (the things that our direct dependencies themselves depend on, and so on recursively.)

We maintain a locked list of all our dependencies (transitive included) so that we can track exactly which version of each dependency appears in a given release. See [here](#) for discussion of why we wanted this for Synapse. We chose to use `poetry` to manage this locked list; see [this comment](#) for the reasoning.

The locked dependencies get included in our "self-contained" releases: namely, our docker images and our debian packages. We also use the locked dependencies in development and our continuous integration.

Separately, our "broad" dependencies—the version ranges specified in `pyproject.toml`—are included as metadata in our "sdists" and "wheels" [uploaded to PyPI](#). Installing from PyPI or from the Synapse source tree directly will *not* use the locked dependencies; instead, they'll pull in the latest version of each package available at install time.

Example dependency

An example may help. We have a broad dependency on `phonenumbers`, as declared in this snippet from `pyproject.toml` [as of Synapse 1.57](#):

```
[tool.poetry.dependencies]
# ...
phonenumbers = ">=8.2.0"
```

In our lockfile this is [pinned](#) to version 8.12.44, even though [newer versions are available](#).

```
[[package]]
name = "phonenumbers"
version = "8.12.44"
description = "Python version of Google's common library for parsing, formatting, storing and validating international phone numbers."
category = "main"
optional = false
python-versions = "*"
```

The lockfile also includes a [cryptographic checksum](#) of the sdists and wheels provided for this version of [phonenumbers](#).

```
[metadata.files]
# ...
phonenumbers = [
    {file = "phonenumbers-8.12.44-py2.py3-none-any.whl", hash =
"sha256:cc1299cf37b309ecab6214297663ab86cb3d64ae37fd5b88e904fe7983a874a6"},

    {file = "phonenumbers-8.12.44.tar.gz", hash =
"sha256:26cf0257d1704fe2f88caff2caabb70d16a877b1e65b6aae51f9fbbe10aa8ce"},

]
```

We can see this pinned version inside the docker image for that release:

```
$ docker pull vectorim/synapse:v1.97.0
...
$ docker run --entrypoint pip vectorim/synapse:v1.97.0 show phonenumbers
Name: phonenumbers
Version: 8.12.44
Summary: Python version of Google's common library for parsing, formatting, storing and validating international phone numbers.
Home-page: https://github.com/daviddrysdale/python-phonenumbers
Author: David Drysdale
Author-email: dmd@lurk.lurk.org
License: Apache License 2.0
Location: /usr/local/lib/python3.9/site-packages
Requires:
Required-by: matrix-synapse
```

Whereas the wheel metadata just contains the broad dependencies:

```
$ cd /tmp
$ wget
https://files.pythonhosted.org/packages/ca/5e/d722d572cc5b3092402b783d6b7185901b4.1.57.0rc1-py3-none-any.whl
...
$ unzip -c matrix_synapse-1.57.0rc1-py3-none-any.whl matrix_synapse-1.57.0rc1.dist-info/METADATA | grep phonenumbers
Requires-Dist: phonenumbers (>=8.2.0)
```

Tooling recommendation: direnv

`direnv` is a tool for activating environments in your shell inside a given directory. Its support for poetry is unofficial (a community wiki recipe only), but works solidly in our experience. We thoroughly recommend it for daily use. To use it:

1. [Install `direnv`](#) - it's likely packaged for your system already.
2. Teach direnv about poetry. The [shell config here](#) needs to be added to `~/.config/direnv/direnvrc` (or more generally `$_XDG_CONFIG_HOME/direnv/direnvrc`).
3. Mark the synapse checkout as a poetry project: `echo layout poetry > .envrc`.
4. Convince yourself that you trust this `.envrc` configuration and project. Then formally confirm this to `direnv` by running `direnv allow`.

Then whenever you navigate to the synapse checkout, you should be able to run e.g. `mypy` instead of `poetry run mypy`; `python` instead of `poetry run python`; and your shell commands will automatically run in the context of poetry's venv, without having to run `poetry shell` beforehand.

How do I...

...reset my venv to the locked environment?

```
poetry install --all-extras --sync
```

...delete everything and start over from scratch?

```
# Stop the current virtualenv if active
$ deactivate

# Remove all of the files from the current environment.
# Don't worry, even though it says "all", this will only
# remove the Poetry virtualenvs for the current project.
$ poetry env remove --all

# Reactivate Poetry shell to create the virtualenv again
$ poetry shell
# Install everything again
$ poetry install --extras all
```

...run a command in the poetry virtualenv?

Use `poetry run cmd args` when you need the python virtualenv context. To avoid typing `poetry run` all the time, you can run `poetry shell` to start a new shell in the poetry virtualenv context. Within `poetry shell`, `python`, `pip`, `mypy`, `trial`, etc. are all run inside the project virtualenv and isolated from the rest of the system.

Roughly speaking, the translation from a traditional virtualenv is:

- `env/bin/activate` -> `poetry shell`, and
- `deactivate` -> close the terminal (Ctrl-D, `exit`, etc.)

See also the direnv recommendation above, which makes `poetry run` and `poetry shell` unnecessary.

...inspect the poetry virtualenv?

Some suggestions:

```
# Current env only
poetry env info
# All envs: this allows you to have e.g. a poetry managed venv for Python 3.7,
# and another for Python 3.10.
poetry env list --full-path
poetry run pip list
```

Note that `poetry show` describes the abstract *lock file* rather than your on-disk environment. With that said, `poetry show --tree` can sometimes be useful.

...add a new dependency?

Either:

- manually update `pyproject.toml`; then `poetry lock --no-update`; or else
- `poetry add packagename`. See `poetry add --help`; note the `--dev`, `--extras` and `--optional` flags in particular.

Include the updated `pyproject.toml` and `poetry.lock` files in your commit.

...remove a dependency?

This is not done often and is untested, but

```
poetry remove packagename
```

ought to do the trick. Alternatively, manually update `pyproject.toml` and `poetry lock --no-update`. Include the updated `pyproject.toml` and `poetry.lock` files in your commit.

...update the version range for an existing dependency?

Best done by manually editing `pyproject.toml`, then `poetry lock --no-update`. Include the updated `pyproject.toml` and `poetry.lock` in your commit.

...update a dependency in the locked environment?

Use

```
poetry update packagename
```

to use the latest version of `packagename` in the locked environment, without affecting the broad dependencies listed in the wheel.

There doesn't seem to be a way to do this whilst locking a *specific* version of `packagename`. We can workaround this (crudely) as follows:

```
poetry add packagename==1.2.3
# This should update pyproject.lock.

# Now undo the changes to pyproject.toml. For example
# git restore pyproject.toml

# Get poetry to recompute the content-hash of pyproject.toml without changing
# the locked package versions.
poetry lock --no-update
```

Either way, include the updated `poetry.lock` file in your commit.

...export a `requirements.txt` file?

```
poetry export --extras all
```

Be wary of bugs in `poetry export` and `pip install -r requirements.txt`.

...build a test wheel?

I usually use

```
poetry run pip install build && poetry run python -m build
```

because `build` is a standardish tool which doesn't require poetry. (It's what we use in CI too). However, you could try `poetry build` too.

...handle a Dependabot pull request?

Synapse uses Dependabot to keep the `poetry.lock` and `Cargo.lock` file up-to-date with the latest releases of our dependencies. The changelog check is omitted for Dependabot PRs; the release script will include them in the changelog.

When reviewing a dependabot PR, ensure that:

- the lockfile changes look reasonable;
- the upstream changelog file (linked in the description) doesn't include any breaking changes;
- continuous integration passes.

In particular, any updates to the type hints (usually packages which start with `types-`) should be safe to merge if linting passes.

Troubleshooting

Check the version of poetry with `poetry --version`.

The minimum version of poetry supported by Synapse is 1.3.2.

It can also be useful to check the version of `poetry-core` in use. If you've installed `poetry` with `pipx`, try `pipx runpip poetry list | grep poetry-core`.

Clear caches: `poetry cache clear --all pypi.`

Poetry caches a bunch of information about packages that isn't readily available from PyPI. (This is what makes poetry seem slow when doing the first `poetry install`.) Try `poetry cache list` and `poetry cache clear --all <name of cache>` to see if that fixes things.

Remove outdated egg-info

Delete the `matrix_synapse.egg-info/` directory from the root of your Synapse install.

This stores some cached information about dependencies and often conflicts with letting Poetry do the right thing.

Try `--verbose` or `--dry-run` arguments.

Sometimes useful to see what poetry's internal logic is.

Cancellation

Sometimes, requests take a long time to service and clients disconnect before Synapse produces a response. To avoid wasting resources, Synapse can cancel request processing for select endpoints marked with the `@cancelable` decorator.

Synapse makes use of Twisted's `Deferred.cancel()` feature to make cancellation work. The `@cancelable` decorator does nothing by itself and merely acts as a flag, signalling to developers and other code alike that a method can be cancelled.

Enabling cancellation for an endpoint

1. Check that the endpoint method, and any `async` functions in its call tree handle cancellation correctly. See [Handling cancellation correctly](#) for a list of things to look out for.
2. Add the `@cancelable` decorator to the `on_GET/POST/PUT/DELETE` method. It's not recommended to make non-`GET` methods cancellable, since cancellation midway through some database updates is less likely to be handled correctly.

Mechanics

There are two stages to cancellation: downward propagation of a `cancel()` call, followed by upwards propagation of a `CancelledError` out of a blocked `await`. Both Twisted and asyncio have a cancellation mechanism.

Method		Exception	in
Twisted	<code>Deferred.cancel()</code>	<code>twisted.internet.defer.CancelledError</code>	Ex
asyncio	<code>Task.cancel()</code>	<code>asyncio.CancelledError</code>	Ba

Deferred.cancel()

When Synapse starts handling a request, it runs the `async` method responsible for handling it using `defer.ensureDeferred`, which returns a `Deferred`. For example:

```
def do_something() -> Deferred[None]:
    ...

@cancellable
async def on_GET() -> Tuple[int, JsonDict]:
    d = make_deferred_yieldable(do_something())
    await d
    return 200, {}

request = defer.ensureDeferred(on_GET())
```

When a client disconnects early, Synapse checks for the presence of the `@cancellable` decorator on `on_GET`. Since `on_GET` is cancellable, `Deferred.cancel()` is called on the `Deferred` from `defer.ensureDeferred`, ie. `request`. Twisted knows which `Deferred` `request` is waiting on and passes the `cancel()` call on to `d`.

The `Deferred` being waited on, `d`, may have its own handling for `cancel()` and pass the call on to other `Deferred`s.

Eventually, a `Deferred` handles the `cancel()` call by resolving itself with a `CancelledError`.

CancelledError

The `CancelledError` gets raised out of the `await` and bubbles up, as per normal Python exception handling.

Handling cancellation correctly

In general, when writing code that might be subject to cancellation, two things must be considered:

- The effect of `CancelledError`s raised out of `await`s.
- The effect of `Deferred`s being `cancel()`ed.

Examples of code that handles cancellation incorrectly include:

- `try-except` blocks which swallow `CancelledError`s.
- Code that shares the same `Deferred`, which may be cancelled, between multiple requests.
- Code that starts some processing that's exempt from cancellation, but uses a logging context from cancellable code. The logging context will be finished upon cancellation, while the uncancelled processing is still using it.

Some common patterns are listed below in more detail.

async function calls

Most functions in Synapse are relatively straightforward from a cancellation standpoint: they don't do anything with `Deferred`s and purely call and `await` other `async` functions.

An `async` function handles cancellation correctly if its own code handles cancellation correctly and all the `async` function it calls handle cancellation correctly. For example:

```
async def do_two_things() -> None:
    check_something()
    await do_something()
    await do_something_else()
```

`do_two_things` handles cancellation correctly if `do_something` and `do_something_else` handle cancellation correctly.

That is, when checking whether a function handles cancellation correctly, its implementation and all its `async` function calls need to be checked, recursively.

As `check_something` is not `async`, it does not need to be checked.

CancelledErrors

Because Twisted's `CancelledError`s are `Exception`s, it's easy to accidentally catch and suppress them. Care must be taken to ensure that `CancelledError`s are allowed to propagate upwards.

<p>Bad:</p> <pre>try: await do_something() except Exception: # `CancelledError` gets # swallowed here. logger.info(...)</pre>	<p>Good:</p> <pre>try: await do_something() except CancelledError: raise except Exception: logger.info(...)</pre>
<p>OK:</p> <pre>try: check_something() # A `CancelledError` won't # ever be raised here. except Exception: logger.info(...)</pre>	<p>Good:</p> <pre>try: await do_something() except ValueError: logger.info(...)</pre>

defer.gatherResults

`defer.gatherResults` produces a `Deferred` which:

- broadcasts `cancel()` calls to every `Deferred` being waited on.
- wraps the first exception it sees in a `FirstError`.

Together, this means that `CancelledError`s will be wrapped in a `FirstError` unless unwrapped. Such `FirstError`s are liable to be swallowed, so they must be unwrapped.

Bad:

```
async def do_something() -> None:
    await make_deferred_yieldable(
        defer.gatherResults([...],
        consumeErrors=True))
    )

try:
    await do_something()
except CancelledError:
    raise
except Exception:
    # `FirstError(CancelledError)` gets swallowed here.
    logger.info(...)
```

Good:

```
async def do_something() -> None:
    await make_deferred_yieldable(
        defer.gatherResults([...],
        consumeErrors=True))
    ).addErrback(unwrapFirstError)

try:
    await do_something()
except CancelledError:
    raise
except Exception:
    logger.info(...)
```

Creation of `Deferreds`

If a function creates a `Deferred`, the effect of cancelling it must be considered. `Deferred`s that get shared are likely to have unintended behaviour when cancelled.

Bad:

Good:

```

cache: Dict[str, Deferred[None]] = {}

def wait_for_room(room_id: str) -> Deferred[None]:
    deferred = cache.get(room_id)
    if deferred is None:
        deferred = Deferred()
        cache[room_id] = deferred
    # `deferred` can have multiple waiters.
    # All of them will observe a
    # `CancelledError` if any one of them is cancelled.
    return make_deferred_yieldable(deferred)

# Request 1
await
wait_for_room("!aAAaaAaaaAAAaAaAA:matrix.org")
# Request 2
await
wait_for_room("!aAAaaAaaaAAAaAaAA:matrix.org")

```

```

cache: Dict[str, Deferred[

def wait_for_room(room_id: deferred = cache.get(r
if deferred is None:
    deferred = Deferred()
    cache[room_id] = deferred
    # `deferred` will never
    # A `CancelledError` with
    # the `await`.
    # `delay_cancellation` return
make_deferred_yieldable(st

# Request 1
await wait_for_room("!aAAaaAaaaAAAaAaAA:matrix.org")
# Request 2
await wait_for_room("!aAAaaAaaaAAAaAaAA:matrix.org")

```

Good:

```

cache: Dict[str, List[Defe

def wait_for_room(room_id: if room_id not in cache:
    cache[room_id] = [ # Each request gets its own.
        deferred = Deferred()
        cache[room_id]].append(deferred)
    return make_deferred_y

# Request 1
await wait_for_room("!aAAaaAaaaAAAaAaAA:matrix.org")
# Request 2
await wait_for_room("!aAAaaAaaaAAAaAaAA:matrix.org")

```

Uncancelled processing

Some `async` functions may kick off some `async` processing which is intentionally protected from cancellation, by `stop_cancellation` or other means. If the `async` processing inherits the logcontext of the request which initiated it, care must be taken to ensure that the logcontext is not finished before the `async` processing completes.

Bad:

Good:

```

cache:
Optional[ObservableDeferred[None]] =
None

async def do_something_else(
    to_resolve: Deferred[None]
) -> None:
    await ...
    logger.info("done!")
    to_resolve.callback(None)

async def do_something() -> None:
    if not cache:
        to_resolve = Deferred()
        cache =
ObservableDeferred(to_resolve)
        # `do_something_else` will never
        be cancelled and
        # can outlive the `request-1`
    logging context.

    run_in_background(do_something_else,
        to_resolve)

    await
make_deferred_yieldable(cache.observe())

with LoggingContext("request-1"):
    await do_something()

```

```

cache:
Optional[ObservableDeferred[None]] =
None

async def do_something_else(
    to_resolve: Deferred[None]
) -> None:
    await ...
    logger.info("done!")
    to_resolve.callback(None)

async def do_something() -> None:
    if not cache:
        to_resolve = Deferred()
        cache =
ObservableDeferred(to_resolve)
        run_in_background(do_something_else,
            to_resolve)
        # We'll wait until
        `do_something_else` is
        # done before raising a
        `CancelledError`.
        await make_deferred_yieldable(
            cache.observe()
        )
    else:
        await
make_deferred_yieldable(cache.observe())

with LoggingContext("request-1"):
    await do_something()

```

OK:

```
cache:  
Optional[ObservableDeferred[None]] =  
None  
  
async def do_something_else(  
    to_resolve: Deferred[None]  
) -> None:  
    await ...  
    logger.info("done!")  
    to_resolve.callback(None)  
  
async def do_something() -> None:  
    if not cache:  
        to_resolve = Deferred()  
        cache =  
ObservableDeferred(to_resolve)  
        # `do_something_else` will get  
its own independent  
        # logging context. `request-1`  
will not count any  
        # metrics from  
`do_something_else`.  
        run_as_background_process(  
            "do_something_else",  
            do_something_else,  
            to_resolve,  
        )  
  
    await  
make_deferred_yieldable(cache.observe())  
  
with LoggingContext("request-1"):  
    await do_something()
```

Log Contexts

To help track the processing of individual requests, synapse uses a '`log_context`' to track which request it is handling at any given moment. This is done via a thread-local variable; a `logging.Filter` is then used to fish the information back out of the thread-local variable and add it to each log record.

Logcontexts are also used for CPU and database accounting, so that we can track which requests were responsible for high CPU use or database activity.

The `synapse.logging.context` module provides facilities for managing the current log context (as well as providing the `LoggingContextFilter` class).

Asynchronous functions make the whole thing complicated, so this document describes how it all works, and how to write code which follows the rules.

In this document, "awaitable" refers to any object which can be `await`ed. In the context of Synapse, that normally means either a coroutine or a Twisted `Deferred`.

Logcontexts without asynchronous code

In the absence of any asynchronous voodoo, things are simple enough. As with any code of this nature, the rule is that our function should leave things as it found them:

```
from synapse.logging import context          # omitted from future snippets

def handle_request(request_id):
    request_context = context.LoggingContext()

    calling_context = context.set_current_context(request_context)
    try:
        request_context.request = request_id
        do_request_handling()
        logger.debug("finished")
    finally:
        context.set_current_context(calling_context)

def do_request_handling():
    logger.debug("phew")  # this will be logged against request_id
```

LoggingContext implements the context management methods, so the above can be written much more succinctly as:

```
def handle_request(request_id):
    with context.LoggingContext() as request_context:
        request_context.request = request_id
        do_request_handling()
        logger.debug("finished")

def do_request_handling():
    logger.debug("phew")
```

Using logcontexts with awaitables

Awaitables break the linear flow of code so that there is no longer a single entry point where we should set the logcontext and a single exit point where we should remove it.

Consider the example above, where `do_request_handling` needs to do some blocking operation, and returns an awaitable:

```
async def handle_request(request_id):
    with context.LoggingContext() as request_context:
        request_context.request = request_id
        await do_request_handling()
        logger.debug("finished")
```

In the above flow:

- The logcontext is set
- `do_request_handling` is called, and returns an awaitable
- `handle_request` awaits the awaitable
- Execution of `handle_request` is suspended

So we have stopped processing the request (and will probably go on to start processing the next), without clearing the logcontext.

To circumvent this problem, synapse code assumes that, wherever you have an awaitable, you will want to `await` it. To that end, wherever functions return awaitables, we adopt the following conventions:

Rules for functions returning awaitables:

-
- If the awaitable is already complete, the function returns with the same logcontext it started with.
 - If the awaitable is incomplete, the function clears the logcontext before returning; when the awaitable completes, it restores the logcontext before running any callbacks.
-

That sounds complicated, but actually it means a lot of code (including the example above) "just works". There are two cases:

- If `do_request_handling` returns a completed awaitable, then the logcontext will still be in place. In this case, execution will continue immediately after the `await`; the "finished" line will be logged against the right context, and the `with` block restores the original context before we return to the caller.
- If the returned awaitable is incomplete, `do_request_handling` clears the logcontext before returning. The logcontext is therefore clear when `handle_request` `awaits` the awaitable.

Once `do_request_handling`'s awaitable completes, it will reinstate the logcontext, before running the second half of `handle_request`, so again the "finished" line will be logged against the right context, and the `with` block restores the original context.

As an aside, it's worth noting that `handle_request` follows our rules

- though that only matters if the caller has its own logcontext which it cares about.

The following sections describe pitfalls and helpful patterns when implementing these rules.

Always await your awaitables

Whenever you get an awaitable back from a function, you should `await` on it as soon as possible. Do not pass go; do not do any logging; do not call any other functions.

```
async def fun():
    logger.debug("starting")
    await do_some_stuff()          # just like this

    coro = more_stuff()
    result = await coro           # also fine, of course

    return result
```

Provided this pattern is followed all the way back up to the callchain to where the logcontext was set, this will make things work out ok: provided `do_some_stuff` and `more_stuff` follow the rules above, then so will `fun`.

It's all too easy to forget to `await`: for instance if we forgot that `do_some_stuff` returned an awaitable, we might plough on regardless. This leads to a mess; it will probably work itself out eventually, but not before a load of stuff has been logged against the wrong context. (Normally, other things will break, more obviously, if you forget to `await`, so this tends not to be a major problem in practice.)

Of course sometimes you need to do something a bit fancier with your awaitable - not all code follows the linear A-then-B-then-C pattern. Notes on implementing more complex patterns are in later sections.

Where you create a new awaitable, make it follow the rules

Most of the time, an awaitable comes from another synapse function. Sometimes, though, we need to make up a new awaitable, or we get an awaitable back from external code. We need to make it follow our rules.

The easy way to do it is by using `context.make_deferred_yieldable`. Suppose we want to implement `sleep`, which returns a deferred which will run its callbacks after a given number of seconds. That might look like:

```
# not a logcontext-rules-compliant function
def get_sleep_deferred(seconds):
    d = defer.Deferred()
    reactor.callLater(seconds, d.callback, None)
    return d
```

That doesn't follow the rules, but we can fix it by calling it through `context.make_deferred_yieldable`:

```
async def sleep(seconds):
    return await context.make_deferred_yieldable(get_sleep_deferred(seconds))
```

Fire-and-forget

Sometimes you want to fire off a chain of execution, but not wait for its result. That might look a bit like this:

```
async def do_request_handling():
    await foreground_operation()

    # *don't* do this
    background_operation()

    logger.debug("Request handling complete")

async def background_operation():
    await first_background_step()
    logger.debug("Completed first step")
    await second_background_step()
    logger.debug("Completed second step")
```

The above code does a couple of steps in the background after `do_request_handling` has finished. The log lines are still logged against the `request_context` logcontext, which may or may not be desirable. There are two big problems with the above, however. The first problem is that, if `background_operation` returns an incomplete awaitable, it will expect its caller to `await` immediately, so will have cleared the logcontext. In this example, that means that 'Request handling complete' will be logged without any context.

The second problem, which is potentially even worse, is that when the awaitable returned by `background_operation` completes, it will restore the original logcontext. There is nothing waiting on that awaitable, so the logcontext will leak into the reactor and possibly get attached to some arbitrary future operation.

There are two potential solutions to this.

One option is to surround the call to `background_operation` with a `PreserveLoggingContext` call. That will reset the logcontext before starting `background_operation` (so the context restored when the deferred completes will be the empty logcontext), and will restore the current logcontext before continuing the foreground process:

```
async def do_request_handling():
    await foreground_operation()

    # start background_operation off in the empty logcontext, to
    # avoid leaking the current context into the reactor.
    with PreserveLoggingContext():
        background_operation()

    # this will now be logged against the request context
    logger.debug("Request handling complete")
```

Obviously that option means that the operations done in `background_operation` would be not be logged against a logcontext (though that might be fixed by setting a different logcontext via a `with LoggingContext(...)` in `background_operation`).

The second option is to use `context.run_in_background`, which wraps a function so that it doesn't reset the logcontext even when it returns an incomplete awaitable, and adds a callback to the returned awaitable to reset the logcontext. In other words, it turns a function that follows the Synapse rules about logcontexts and awaitables into one which behaves more like an external function --- the opposite operation to that described in the previous section. It can be used like this:

```
async def do_request_handling():
    await foreground_operation()

    context.run_in_background(background_operation)

    # this will now be logged against the request context
    logger.debug("Request handling complete")
```

Passing synapse deferreds into third-party functions

A typical example of this is where we want to collect together two or more awaitables via `defer.gatherResults`:

```
a1 = operation1()
a2 = operation2()
a3 = defer.gatherResults([a1, a2])
```

This is really a variation of the fire-and-forget problem above, in that we are firing off `a1` and `a2` without awaiting on them. The difference is that we now have third-party code attached to their callbacks. Anyway either technique given in the [Fire-and-forget](#) section will work.

Of course, the new awaitable returned by `gather` needs to be wrapped in order to make it follow the logcontext rules before we can yield it, as described in [Where you create a new awaitable, make it follow the rules](#).

So, option one: reset the logcontext before starting the operations to be gathered:

```
async def do_request_handling():
    with PreserveLoggingContext():
        a1 = operation1()
        a2 = operation2()
        result = await defer.gatherResults([a1, a2])
```

In this case particularly, though, option two, of using `context.run_in_background` almost certainly makes more sense, so that `operation1` and `operation2` are both logged against the original logcontext. This looks like:

```
async def do_request_handling():
    a1 = context.run_in_background(operation1)
    a2 = context.run_in_background(operation2)

    result = await make_deferred_yieldable(defer.gatherResults([a1, a2]))
```

A note on garbage-collection of awaitable chains

It turns out that our logcontext rules do not play nicely with awaitable chains which get orphaned and garbage-collected.

Imagine we have some code that looks like this:

```

listener_queue = []

def on_something_interesting():
    for d in listener_queue:
        d.callback("foo")

async def await_something_interesting():
    new_awaitable = defer.Deferred()
    listener_queue.append(new_awaitable)

    with PreserveLoggingContext():
        await new_awaitable

```

Obviously, the idea here is that we have a bunch of things which are waiting for an event. (It's just an example of the problem here, but a relatively common one.)

Now let's imagine two further things happen. First of all, whatever was waiting for the interesting thing goes away. (Perhaps the request times out, or something *even more* interesting happens.)

Secondly, let's suppose that we decide that the interesting thing is never going to happen, and we reset the listener queue:

```

def reset_listener_queue():
    listener_queue.clear()

```

So, both ends of the awaitable chain have now dropped their references, and the awaitable chain is now orphaned, and will be garbage-collected at some point. Note that

`await_something_interesting` is a coroutine, which Python implements as a generator function. When Python garbage-collects generator functions, it gives them a chance to clean up by making the `await` (or `yield`) raise a `GeneratorExit` exception. In our case, that means that the `__exit__` handler of `PreserveLoggingContext` will carefully restore the request context, but there is now nothing waiting for its return, so the request context is never cleared.

To reiterate, this problem only arises when *both* ends of a awaitable chain are dropped. Dropping the the reference to an awaitable you're supposed to be awaiting is bad practice, so this doesn't actually happen too much. Unfortunately, when it does happen, it will lead to leaked logcontexts which are incredibly hard to track down.

Replication Architecture

Motivation

We'd like to be able to split some of the work that synapse does into multiple python processes. In theory multiple synapse processes could share a single postgresql database and we'd scale up by running more synapse processes. However much of synapse assumes that only one process is interacting with the database, both for assigning unique identifiers when inserting into tables, notifying components about new updates, and for invalidating its caches.

So running multiple copies of the current code isn't an option. One way to run multiple processes would be to have a single writer process and multiple reader processes connected to the same database. In order to do this we'd need a way for the reader process to invalidate its in-memory caches when an update happens on the writer. One way to do this is for the writer to present an append-only log of updates which the readers can consume to invalidate their caches and to push updates to listening clients or pushers.

Synapse already stores much of its data as an append-only log so that it can correctly respond to `/sync` requests so the amount of code changes needed to expose the append-only log to the readers should be fairly minimal.

Architecture

The Replication Protocol

See [the TCP replication documentation](#).

The TCP Replication Module

Information about how the tcp replication module is structured, including how the classes interact, can be found in `synapse/replication/tcp/__init__.py`

Streams

Synapse has a concept of "streams", which are roughly described in [id_generators.py](#). Generally speaking, streams are a series of notifications that something in Synapse's database has changed that the application might need to respond to. For example:

- The events stream reports new events (PDUs) that Synapse creates, or that Synapse accepts from another homeserver.
- The account data stream reports changes to users' [account data](#).
- The to-device stream reports when a device has a new [to-device message](#).

See [synapse.replication.tcp.streams](#) for the full list of streams.

It is very helpful to understand the streams mechanism when working on any part of Synapse that needs to respond to changes—especially if those changes are made by different workers. To that end, let's describe streams formally, paraphrasing from the docstring of [AbstractStreamIdGenerator](#).

Definition

A stream is an append-only log $T_1, T_2, \dots, T_n, \dots$ of facts¹ which grows over time. Only "writers" can add facts to a stream, and there may be multiple writers.

Each fact has an ID, called its "stream ID". Readers should only process facts in ascending stream ID order.

Roughly speaking, each stream is backed by a database table. It should have a [stream_id](#) (or similar) bigint column holding stream IDs, plus additional columns as necessary to describe the fact. Typically, a fact is expressed with a single row in its backing table.² Within a stream, no two facts may have the same `stream_id`.

Aside. Some additional notes on streams' backing tables.

1. Rich would like to [ditch the backing tables](#).
2. The backing tables may have other uses. > For example, the events table serves backs the events stream, and is read when processing new events. > But old rows are read from the table all the time, whenever Synapse needs to lookup some facts about an event.
3. Rich suspects that sometimes the stream is backed by multiple tables, so the stream proper is the union of those tables.

Stream writers can "reserve" a stream ID, and then later mark it as having been completed. Stream writers need to track the completion of each stream fact. In the happy case, completion means a fact has been written to the stream table. But unhappy cases (e.g. transaction rollback due to an error) also count as completion. Once completed, the rows written with that stream ID are fixed, and no new rows will be inserted with that ID.

Current stream ID

For any given stream reader (including writers themselves), we may define a per-writer current stream ID:

A current stream ID for a writer W is the largest stream ID such that all transactions added by W with equal or smaller ID have completed.

Similarly, there is a "linear" notion of current stream ID:

A "linear" current stream ID is the largest stream ID such that all facts (added by any writer) with equal or smaller ID have completed.

Because different stream readers A and B learn about new facts at different times, A and B may disagree about current stream IDs. Put differently: we should think of stream readers as being independent of each other, proceeding through a stream of facts at different rates.

The above definition does not give a unique current stream ID, in fact there can be a range of current stream IDs. Synapse uses both the minimum and maximum IDs for different purposes. Most often the maximum is used, as it's generally beneficial for workers to advance their IDs as soon as possible. However, the minimum is used in situations where e.g. another worker is going to wait until the stream advances past a position.

NB. For both senses of "current", that if a writer opens a transaction that never completes, the current stream ID will never advance beyond that writer's last written stream ID.

For single-writer streams, the per-writer current ID and the linear current ID are the same. Both senses of current ID are monotonic, but they may "skip" or jump over IDs because facts complete out of order.

Example. Consider a single-writer stream which is initially at ID 1.

Action	Current stream ID	Notes
	1	
Reserve 2	1	

Action	Current stream ID	Notes
Reserve 3	1	
Complete 3	1	current ID unchanged, waiting for 2 to complete
Complete 2	3	current ID jumps from 1 -> 3
Reserve 4	3	
Reserve 5	3	
Reserve 6	3	
Complete 5	3	
Complete 4	5	current ID jumps 3->5, even though 6 is pending
Complete 6	6	

Multi-writer streams

There are two ways to view a multi-writer stream.

1. Treat it as a collection of distinct single-writer streams, one for each writer.
2. Treat it as a single stream.

The single stream (option 2) is conceptually simpler, and easier to represent (a single stream id). However, it requires each reader to know about the entire set of writers, to ensure that readers don't erroneously advance their current stream position too early and miss a fact from an unknown writer. In contrast, multiple parallel streams (option 1) are more complex, requiring more state to represent (map from writer to stream id). The payoff for doing so is that readers can "peek" ahead to facts that completed on one writer no matter the state of the others, reducing latency.

Note that a multi-writer stream can be viewed in both ways. For example, the events stream is treated as multiple single-writer streams (option 1) by the sync handler, so that events are sent to clients as soon as possible. But the background process that works through events treats them as a single linear stream.

Another useful example is the cache invalidation stream. The facts this stream holds are instructions to "you should now invalidate these cache entries". We only ever treat this as a multiple single-writer streams as there is no important ordering between cache invalidations. (Invalidations are self-contained facts; and the invalidations commute/are idempotent).

Writing to streams

Writers need to track:

- track their current position (i.e. its own per-writer stream ID).
- their facts currently awaiting completion.

At startup,

- the current position of that writer can be found by querying the database (which suggests that facts need to be written to the database atomically, in a transaction); and
- there are no facts awaiting completion.

To reserve a stream ID, call `nextval` on the appropriate postgres sequence.

To write a fact to the stream: insert the appropriate rows to the appropriate backing table.

To complete a fact, first remove it from your map of facts currently awaiting completion. Then, if no earlier fact is awaiting completion, the writer can advance its current position in that stream. Upon doing so it should emit an `RDATA` message³, once for every fact between the old and the new stream ID.

Subscribing to streams

Readers need to track the current position of every writer.

At startup, they can find this by contacting each writer with a `REPLICATE` message, requesting that all writers reply describing their current position in their streams. Writers reply with a `POSITION` message.

To learn about new facts, readers should listen for `RDATA` messages and process them to respond to the new fact. The `RDATA` itself is not a self-contained representation of the fact; readers will have to query the stream tables for the full details. Readers must also advance their record of the writer's current position for that stream.

Summary

In a nutshell: we have an append-only log with a "buffer/scratchpad" at the end where we have to wait for the sequence to be linear and contiguous.

¹ we use the word *fact* here for two reasons. Firstly, the word "event" is already heavily overloaded (PDUs, EDUs, account data, ...) and we don't need to make that worse. Secondly, "fact" emphasises that the things we append to a stream cannot change after the fact.

² A fact might be expressed with 0 rows, e.g. if we opened a transaction to persist an event, but failed and rolled the transaction back before marking the fact as completed. In principle a fact might be expressed with 2 or more rows; if so, each of those rows should share the fact's stream ID.

³ This communication used to happen directly with the writers [over TCP](#); nowadays it's done via Redis's Pubsub.

TCP Replication

Motivation

Previously the workers used an HTTP long poll mechanism to get updates from the master, which had the problem of causing a lot of duplicate work on the server. This TCP protocol replaces those APIs with the aim of increased efficiency.

Overview

The protocol is based on fire and forget, line based commands. An example flow would be (where '>' indicates master to worker and '<' worker to master flows):

```
> SERVER example.com
< REPLICATE
> POSITION events master 53 53
> RDATA events master 54 ["$foo1:bar.com", ...]
> RDATA events master 55 ["$foo4:bar.com", ...]
```

The example shows the server accepting a new connection and sending its identity with the **SERVER** command, followed by the client server to respond with the position of all streams. The server then periodically sends **RDATA** commands which have the format **RDATA <stream_name> <instance_name> <token> <row>**, where the format of **<row>** is defined by the individual streams. The **<instance_name>** is the name of the Synapse process that generated the data (usually "master"). We expect an RDATA for every row in the DB.

Error reporting happens by either the client or server sending an **ERROR** command, and usually the connection will be closed.

Since the protocol is a simple line based, its possible to manually connect to the server using a tool like netcat. A few things should be noted when manually using the protocol:

- The federation stream is only available if federation sending has been disabled on the main process.
- The server will only time connections out that have sent a **PING** command. If a ping is sent then the connection will be closed if no further commands are received within 15s. Both the client and server protocol implementations will send an initial PING on connection and ensure at least one command every 5s is sent (not necessarily **PING**).
- **RDATA** commands *usually* include a numeric token, however if the stream has multiple rows to replicate per token the server will send multiple **RDATA** commands, with all but

the last having a token of `batch`. See the documentation on `commands.RdataCommand` for further details.

Architecture

The basic structure of the protocol is line based, where the initial word of each line specifies the command. The rest of the line is parsed based on the command. For example, the RDATA command is defined as:

```
RDATA <stream_name> <instance_name> <token> <row_json>
```

(Note that `<row_json>` may contain spaces, but cannot contain newlines.)

Blank lines are ignored.

Keep alives

Both sides are expected to send at least one command every 5s or so, and should send a `PING` command if necessary. If either side do not receive a command within e.g. 15s then the connection should be closed.

Because the server may be connected to manually using e.g. netcat, the timeouts aren't enabled until an initial `PING` command is seen. Both the client and server implementations below send a `PING` command immediately on connection to ensure the timeouts are enabled.

This ensures that both sides can quickly realize if the tcp connection has gone and handle the situation appropriately.

Start up

When a new connection is made, the server:

- Sends a `SERVER` command, which includes the identity of the server, allowing the client to detect if its connected to the expected server
- Sends a `PING` command as above, to enable the client to time out connections promptly.

The client:

- Sends a `NAME` command, allowing the server to associate a human friendly name with the connection. This is optional.
- Sends a `PING` as above

- Sends a **REPLICATE** to get the current position of all streams.
- On receipt of a **SERVER** command, checks that the server name matches the expected server name.

Error handling

If either side detects an error it can send an **ERROR** command and close the connection.

If the client side loses the connection to the server it should reconnect, following the steps above.

Congestion

If the server sends messages faster than the client can consume them the server will first buffer a (fairly large) number of commands and then disconnect the client. This ensures that we don't queue up an unbounded number of commands in memory and gives us a potential opportunity to squawk loudly. When/if the client recovers it can reconnect to the server and ask for missed messages.

Reliability

In general the replication stream should be considered an unreliable transport since e.g. commands are not resent if the connection disappears.

The exception to that are the replication streams, i.e. RDATA commands, since these include tokens which can be used to restart the stream on connection errors.

The client should keep track of the token in the last RDATA command received for each stream so that on reconnection it can start streaming from the correct place. Note: not all RDATA have valid tokens due to batching. See **RdataCommand** for more details.

Example

An example interaction is shown below. Each line is prefixed with '>' or '<' to indicate which side is sending, these are *not* included on the wire:

```

* connection established *
> SERVER localhost:8823
> PING 1490197665618
< NAME synapse.app.appservice
< PING 1490197665618
< REPLICATE
> POSITION events master 1 1
> POSITION backfill master 1 1
> POSITION caches master 1 1
> RDATA caches master 2 ["get_user_by_id",["@01register-
user:localhost:8823"],1490197670513]
> RDATA events master 14 ["$149019767112v0Hxz:localhost:8823",
  "!AFDCvgApUmpdfVjIXm:localhost:8823","m.room.guest_access","",null]
< PING 1490197675618
> ERROR server stopping
* connection closed by server *

```

The **POSITION** command sent by the server is used to set the clients position without needing to send data with the **RDATA** command.

An example of a batched set of **RDATA** is:

```

> RDATA caches master batch ["get_user_by_id",
  ["@test:localhost:8823"],1490197670513]
> RDATA caches master batch ["get_user_by_id",
  ["@test2:localhost:8823"],1490197670513]
> RDATA caches master batch ["get_user_by_id",
  ["@test3:localhost:8823"],1490197670513]
> RDATA caches master 54 ["get_user_by_id",
  ["@test4:localhost:8823"],1490197670513]

```

In this case the client shouldn't advance their caches token until it sees the the last **RDATA**.

List of commands

The list of valid commands, with which side can send it: server (S) or client (C):

SERVER (S)

Sent at the start to identify which server the client is talking to

RDATA (S)

A single update in a stream

POSITION (S)

On receipt of a POSITION command clients should check if they have missed any updates, and if so then fetch them out of band. Sent in response to a REPLICATE command (but can happen at any time).

The POSITION command includes the source of the stream. Currently all streams are written by a single process (usually "master"). If fetching missing updates via HTTP API, rather than via the DB, then processes should make the request to the appropriate process.

Two positions are included, the "new" position and the last position sent respectively. This allows servers to tell instances that the positions have advanced but no data has been written, without clients needlessly checking to see if they have missed any updates. Instances will only fetch stuff if there is a gap between their current position and the given last position.

ERROR (S, C)

There was an error

PING (S, C)

Sent periodically to ensure the connection is still alive

NAME (C)

Sent at the start by client to inform the server who they are

REPLICATE (C)

Asks the server for the current position of all streams.

USER_SYNC (C)

A user has started or stopped syncing on this process.

CLEAR_USER_SYNC (C)

The server should clear all associated user sync data from the worker.

This is used when a worker is shutting down.

FEDERATION_ACK (C)

Acknowledge receipt of some federation data

REMOTE_SERVER_UP (S, C)

Inform other processes that a remote server may have come back online.

See `synapse/replication/tcp/commands.py` for a detailed description and the format of each command.

Cache Invalidations Stream

The cache invalidation stream is used to inform workers when they need to invalidate any of their caches in the data store. This is done by streaming all cache invalidations done on master down to the workers, assuming that any caches on the workers also exist on the master.

Each individual cache invalidation results in a row being sent down replication, which includes the cache name (the name of the function) and the key to invalidate. For example:

```
> RDATA caches master 550953771 ["get_user_by_id", ["@bob@example.com"], 1550574873251]
```

Alternatively, an entire cache can be invalidated by sending down a `null` instead of the key. For example:

```
> RDATA caches master 550953772 ["get_user_by_id", null, 1550574873252]
```

However, there are times when a number of caches need to be invalidated at the same time with the same key. To reduce traffic we batch those invalidations into a single poke by defining a special cache name that workers understand to mean to expand to invalidate the correct caches.

Currently the special cache names are declared in `synapse/storage/_base.py` and are:

1. `cs_cache_fake` — invalidates caches that depend on the current state

How do faster joins work?

This is a work-in-progress set of notes with two goals:

- act as a reference, explaining how Synapse implements faster joins; and
- record the rationale behind our choices.

See also [MSC3902](#).

The key idea is described by [MSC3706](#). This allows servers to request a lightweight response to the federation `/send_join` endpoint. This is called a **faster join**, also known as a **partial join**. In these notes we'll usually use the word "partial" as it matches the database schema.

Overview: processing events in a partially-joined room

The response to a partial join consists of

- the requested join event **J**,
- a list of the servers in the room (according to the state before **J**),
- a subset of the state of the room before **J**,
- the full auth chain of that state subset.

Synapse marks the room as partially joined by adding a row to the database table `partial_state_rooms`. It also marks the join event **J** as "partially stated", meaning that we have neither received nor computed the full state before/after **J**. This is done by adding a row to `partial_state_events`.

► DB schema

While partially joined to a room, Synapse receives events **E** from remote homeservers as normal, and can create events at the request of its local users. However, we run into trouble when we enforce the [checks on an event](#).

1. Is a valid event, otherwise it is dropped. For an event to be valid, it must contain a `room_id`, and it must comply with the event format of that room version.
2. Passes signature checks, otherwise it is dropped.
3. Passes hash checks, otherwise it is redacted before being processed further.
4. Passes authorization rules based on the event's auth events, otherwise it is rejected.
5. **Passes authorization rules based on the state before the event, otherwise it is rejected.**

6. Passes authorization rules based on the current state of the room, otherwise it is "soft failed".

We can enforce checks 1--4 without any problems. But we cannot enforce checks 5 or 6 with complete certainty, since Synapse does not know the full state before `E`, nor that of the room.

Partial state

Instead, we make a best-effort approximation. While the room is considered partially joined, Synapse tracks the "partial state" before events. This works in a similar way as regular state:

- The partial state before `J` is that given to us by the partial join response.
- The partial state before an event `E` is the resolution of the partial states after each of `E`'s `prev_events`.
- If `E` is rejected or a message event, the partial state after `E` is the partial state before `E`.
- Otherwise, the partial state after `E` is the partial state before `E`, plus `E` itself.

More concisely, partial state propagates just like full state; the only difference is that we "seed" it with an incomplete initial state. Synapse records that we have only calculated partial state for this event with a row in `partial_state_events`.

While the room remains partially stated, check 5 on incoming events to that room becomes:

5. Passes authorization rules based on **the resolution between the partial state before `E` and `E`'s auth events**. If the event fails to pass authorization rules, it is rejected.

Additionally, check 6 is deleted: no soft-failures are enforced.

While partially joined, the current partial state of the room is defined as the resolution across the partial states after all forward extremities in the room.

Remark. Events with partial state are *not* considered [outliers](#).

Approximation error

Using partial state means the auth checks can fail in a few different ways¹.

¹ Is this exhaustive?

- We may erroneously accept an incoming event in check 5 based on partial state when it would have been rejected based on full state, or vice versa.
- This means that an event could erroneously be added to the current partial state of the room when it would not be present in the full state of the room, or vice versa.
- Additionally, we may have skipped soft-failing an event that would have been soft-failed based on full state.

(Note that the discrepancies described in the last two bullets are user-visible.)

This means that we have to be very careful when we want to lookup pieces of room state in a partially-joined room. Our approximation of the state may be incorrect or missing. But we can make some educated guesses. If

- our partial state is likely to be correct, or
- the consequences of our partial state being incorrect are minor,

then we proceed as normal, and let the resync process fix up any mistakes (see below).

When is our partial state likely to be correct?

- It's more accurate the closer we are to the partial join event. (So we should ideally complete the resync as soon as possible.)
- Non-member events: we will have received them as part of the partial join response, if they were part of the room state at that point. We may incorrectly accept or reject updates to that state (at first because we lack remote membership information; later because of compounding errors), so these can become incorrect over time.
- Local members' memberships: we are the only ones who can create join and knock events for our users. We can't be completely confident in the correctness of bans, invites and kicks from other homeservers, but the resync process should correct any mistakes.
- Remote members' memberships: we did not receive these in the /send_join response, so we have essentially no idea if these are correct or not.

In short, we deem it acceptable to trust the partial state for non-membership and local membership events. For remote membership events, we wait for the resync to complete, at which point we have the full state of the room and can proceed as normal.

Fixing the approximation with a resync

The partial-state approximation is only a temporary affair. In the background, synapse beings a "resync" process. This is a continuous loop, starting at the partial join event and proceeding downwards through the event graph. For each **E** seen in the room since partial join, Synapse will fetch

- the event ids in the state of the room before **E**, via [/state_ids](#);
- the event ids in the full auth chain of **E**, included in the [/state_ids](#) response; and

- any events from the previous two bullets that Synapse hasn't persisted, via `/state`.

This means Synapse has (or can compute) the full state before **E**, which allows Synapse to properly authorise or reject **E**. At this point, the event is considered to have "full state" rather than "partial state". We record this by removing **E** from the `partial_state_events` table.

[**TODO:** Does Synapse persist a new state group for the full state before **E**, or do we alter the (partial-)state group in-place? Are state groups ever marked as partially-stated?]

This scheme means it is possible for us to have accepted and sent an event to clients, only to reject it during the resync. From a client's perspective, the effect is similar to a retroactive state change due to state resolution---i.e. a "state reset".²

² Clients should refresh caches to detect such a change. Rumour has it that sliding sync will fix this.

When all events since the join **J** have been fully-stated, the room resync process is complete. We record this by removing the room from `partial_state_rooms`.

Faster joins on workers

For the time being, the resync process happens on the master worker. A new replication stream `un_partial_stated_room` is added. Whenever a resync completes and a partial-state room becomes fully stated, a new message is sent into that stream containing the room ID.

Notes on specific cases

NB. The notes below are rough. Some of them are hidden under `<details>` disclosures because they have yet to be implemented in mainline Synapse.

Creating events during a partial join

When sending out messages during a partial join, we assume our partial state is accurate and proceed as normal. For this to have any hope of succeeding at all, our partial state must contain an entry for each of the (type, state key) pairs [specified by the auth rules](#):

- `m.room.create`
- `m.room.join_rules`

- `m.room.power_levels`
- `m.room.third_party_invite`
- `m.room.member`

The first four of these should be present in the state before `J` that is given to us in the partial join response; only membership events are omitted. In order for us to consider the user joined, we must have their membership event. That means the only possible omission is the target's membership in an invite, kick or ban.

The worst possibility is that we locally invite someone who is banned according to the full state, because we lack their ban in our current partial state. The rest of the federation---at least, those who are fully joined---should correctly enforce the [membership transition constraints](#). So any the erroneous invite should be ignored by fully-joined homeservers and resolved by the resync for partially-joined homeservers.

In more generality, there are two problems we're worrying about here:

- We might create an event that is valid under our partial state, only to later find out that is actually invalid according to the full state.
- Or: we might refuse to create an event that is invalid under our partial state, even though it would be perfectly valid under the full state.

However we expect such problems to be unlikely in practise, because

- We trust that the room has sensible power levels, e.g. that bad actors with high power levels are demoted before their ban.
- We trust that the resident server provides us up-to-date power levels, join rules, etc.
- State changes in rooms are relatively infrequent, and the resync period is relatively quick.

Sending out the event over federation

TODO: needs prose fleshing out.

Normally: send out in a fed txn to all HSes in the room. We only know that some HSes were in the room at some point. What do. Send it out to the list of servers from the first join. **TODO** what do we do here if we have full state? If the prev event was created by us, we can risk sending it to the wrong HS. (Motivation: privacy concern of the content. Not such a big deal for a public room or an encrypted room. But non-encrypted invite-only...) But don't want to send out sensitive data in other HS's events in this way.

Suppose we discover after resync that we shouldn't have sent out one our events (not a `prev_event`) to a target HS. Not much we can do. What about if we didn't send them an event but shouldn't've? E.g. what if someone joined from a new HS shortly after you did? We wouldn't talk to them. Could imagine sending out the "Missed" events after the resync but... painful to work out what they should have seen if they joined/left. Instead, just send them the latest event (if they're still in the room after resync) and let them backfill.(?)

- Don't do this currently.
- If anyone who has received our messages sends a message to a HS we missed, they can backfill our messages
- Gap: rooms which are infrequently used and take a long time to resync.

Joining after a partial join

NB. Not yet implemented.

► Détails

Leaving (and kicks and bans) after a partial join

NB. Not yet implemented.

► Détails

Internal Documentation

This section covers implementation documentation for various parts of Synapse.

If a developer is planning to make a change to a feature of Synapse, it can be useful for general documentation of how that feature is implemented to be available. This saves the developer time in place of needing to understand how the feature works by reading the code.

Documentation that would be more useful for the perspective of a system administrator, rather than a developer who's intending to change to code, should instead be placed under the Usage section of the documentation.

How to test SAML as a developer without a server

<https://fujifish.github.io/samling/samling.html> (<https://github.com/fujifish/samling>) is a great resource for being able to tinker with the SAML options within Synapse without needing to deploy and configure a complicated software stack.

To make Synapse (and therefore Element) use it:

1. Use the `samling.html` URL above or deploy your own and visit the IdP Metadata tab.
2. Copy the XML to your clipboard.
3. On your Synapse server, create a new file `samling.xml` next to your `homeserver.yaml` with the XML from step 2 as the contents.
4. Edit your `homeserver.yaml` to include:

```
saml2_config:
  sp_config:
    allow_unknown_attributes: true # Works around a bug with AVA Hashes:
    https://github.com/IdentityPython/pysaml2/issues/388
    metadata:
      local: ["samling.xml"]
```

5. Ensure that your `homeserver.yaml` has a setting for `public_baseurl`:

```
public_baseurl: http://localhost:8080/
```

6. Run `apt-get install xmlsec1` and `pip install --upgrade --force 'pysaml2>=4.5.0'` to ensure the dependencies are installed and ready to go.
7. Restart Synapse.

Then in Element:

1. Visit the login page and point Element towards your homeserver using the `public_baseurl` above.
2. Click the Single Sign-On button.
3. On the `samling` page, enter a Name Identifier and add a SAML Attribute for `uid=your_localpart`. The response must also be signed.
4. Click "Next".
5. Click "Post Response" (change nothing).
6. You should be logged in.

If you try and repeat this process, you may be automatically logged in using the information you gave previously. To fix this, open your developer console (`F12` or `Ctrl+Shift+I`) while

on the samling page and clear the site data. In Chrome, this will be a button on the Application tab.

How to test CAS as a developer without a server

The [django-mama-cas](#) project is an easy to run CAS implementation built on top of Django.

Prerequisites

1. Create a new virtualenv: `python3 -m venv <your virtualenv>`
2. Activate your virtualenv: `source /path/to/your/virtualenv/bin/activate`
3. Install Django and django-mama-cas:

```
python -m pip install "django<3" "django-mama-cas==2.4.0"
```

4. Create a Django project in the current directory:

```
django-admin startproject cas_test .
```

5. Follow the [install directions](#) for django-mama-cas
6. Setup the SQLite database: `python manage.py migrate`
7. Create a user:

```
python manage.py createsuperuser
```

1. Use whatever you want as the username and password.
2. Leave the other fields blank.
8. Use the built-in Django test server to serve the CAS endpoints on port 8000:

```
python manage.py runserver
```

You should now have a Django project configured to serve CAS authentication with a single user created.

Configure Synapse (and Element) to use CAS

1. Modify your `homeserver.yaml` to enable CAS and point it to your locally running Django test server:

```
cas_config:
  enabled: true
  server_url: "http://localhost:8000"
  service_url: "http://localhost:8081"
  #displayname_attribute: name
  #required_attributes:
  #    name: value
```

2. Restart Synapse.

Note that the above configuration assumes the homeserver is running on port 8081 and that the CAS server is on port 8000, both on localhost.

Testing the configuration

Then in Element:

1. Visit the login page with a Element pointing at your homeserver.
2. Click the Single Sign-On button.
3. Login using the credentials created with `createsuperuser`.
4. You should be logged in.

If you want to repeat this process you'll need to manually logout first:

1. `http://localhost:8000/admin/`
2. Click "logout" in the top right.

Room DAG concepts

Edges

The word "edge" comes from graph theory lingo. An edge is just a connection between two events. In Synapse, we connect events by specifying their `prev_events`. A subsequent event points back at a previous event.

```
A (oldest) ----- B ----- C (most recent)
```

Depth and stream ordering

Events are normally sorted by `(topological_ordering, stream_ordering)` where `topological_ordering` is just `depth`. In other words, we first sort by `depth` and then tie-break based on `stream_ordering`. `depth` is incremented as new messages are added to the DAG. Normally, `stream_ordering` is an auto incrementing integer, but backfilled events start with `stream_ordering=-1` and decrement.

- Incremental `/sync?since=xxx` returns things in the order they arrive at the server (`stream_ordering`).
- Initial `/sync`, `/messages` (and `/backfill` in the federation API) return them in the order determined by the event graph `(topological_ordering, stream_ordering)`.

The general idea is that, if you're following a room in real-time (i.e. `/sync`), you probably want to see the messages as they arrive at your server, rather than skipping any that arrived late; whereas if you're looking at a historical section of timeline (i.e. `/messages`), you want to see the best representation of the state of the room as others were seeing it at the time.

Outliers

We mark an event as an `outlier` when we haven't figured out the state for the room at that point in the DAG yet. They are "floating" events that we haven't yet correlated to the DAG.

Outliers typically arise when we fetch the auth chain or state for a given event. When that happens, we just grab the events in the state/auth chain, without calculating the state at those events, or backfilling their `prev_events`. Since we don't have the state at any events fetched in that way, we mark them as outliers.

So, typically, we won't have the `prev_events` of an `outlier` in the database, (though it's entirely possible that we *might* have them for some other reason). Other things that make outliers different from regular events:

- We don't have state for them, so there should be no entry in `event_to_state_groups` for an outlier. (In practice this isn't always the case, though I'm not sure why: see <https://github.com/matrix-org/synapse/issues/12201>).
- We don't record entries for them in the `event_edges`, `event_forward_extremities` or `event_backward_extremities` tables.

Since outliers are not tied into the DAG, they do not normally form part of the timeline sent down to clients via `/sync` or `/messages`; however there is an exception:

Out-of-band membership events

A special case of outlier events are some membership events for federated rooms that we aren't full members of. For example:

- invites received over federation, before we join the room
- *rejections* for said invites
- knock events for rooms that we would like to join but have not yet joined.

In all the above cases, we don't have the state for the room, which is why they are treated as outliers. They are a bit special though, in that they are proactively sent to clients via `/sync`.

Forward extremity

Most-recent-in-time events in the DAG which are not referenced by any other events' `prev_events` yet. (In this definition, outliers, rejected events, and soft-failed events don't count.)

The forward extremities of a room (or at least, a subset of them, if there are more than ten) are used as the `prev_events` when the next event is sent.

The "current state" of a room (ie: the state which would be used if we generated a new event) is, therefore, the resolution of the room states at each of the forward extremities.

Backward extremity

The current marker of where we have backfilled up to and will generally be the `prev_events` of the oldest-in-time events we have in the DAG. This gives a starting point

when backfilling history.

Note that, unlike forward extremities, we typically don't have any backward extremity events themselves in the database - or, if we do, they will be "outliers" (see above). Either way, we don't expect to have the room state at a backward extremity.

When we persist a non-outlier event, if it was previously a backward extremity, we clear it as a backward extremity and set all of its `prev_events` as the new backward extremities if they aren't already persisted as non-outliers. This therefore keeps the backward extremities up-to-date.

State groups

For every non-outlier event we need to know the state at that event. Instead of storing the full state for each event in the DB (i.e. a `event_id -> state` mapping), which is *very* space inefficient when state doesn't change, we instead assign each different set of state a "state group" and then have mappings of `event_id -> state_group` and `state_group -> state`.

Stage group edges

TODO: `state_group_edges` is a further optimization... notes from @Azrenbeth, <https://pastebin.com/seUGVGeT>

Auth Chain Difference Algorithm

The auth chain difference algorithm is used by V2 state resolution, where a naive implementation can be a significant source of CPU and DB usage.

Definitions

A *state set* is a set of state events; e.g. the input of a state resolution algorithm is a collection of state sets.

The *auth chain* of a set of events are all the events' auth events and *their* auth events, recursively (i.e. the events reachable by walking the graph induced by an event's auth events links).

The *auth chain difference* of a collection of state sets is the union minus the intersection of the sets of auth chains corresponding to the state sets, i.e an event is in the auth chain difference if it is reachable by walking the auth event graph from at least one of the state sets but not from *all* of the state sets.

Breadth First Walk Algorithm

A way of calculating the auth chain difference without calculating the full auth chains for each state set is to do a parallel breadth first walk (ordered by depth) of each state set's auth chain. By tracking which events are reachable from each state set we can finish early if every pending event is reachable from every state set.

This can work well for state sets that have a small auth chain difference, but can be very inefficient for larger differences. However, this algorithm is still used if we don't have a chain cover index for the room (e.g. because we're in the process of indexing it).

Chain Cover Index

Synapse computes auth chain differences by pre-computing a "chain cover" index for the auth chain in a room, allowing us to efficiently make reachability queries like "is event **A** in the auth chain of event **B**?". We could do this with an index that tracks all pairs **(A, B)** such that **A** is in the auth chain of **B**. However, this would be prohibitively large, scaling poorly as the room accumulates more state events.

Instead, we break down the graph into *chains*. A chain is a subset of a DAG with the following property: for any pair of events E and F in the chain, the chain contains a path $E \rightarrow F$ or a path $F \rightarrow E$. This forces a chain to be linear (without forks), e.g. $E \rightarrow F \rightarrow G \rightarrow \dots \rightarrow H$. Each event in the chain is given a *sequence number* local to that chain. The oldest event E in the chain has sequence number 1. If E has a child F in the chain, then F has sequence number 2. If E has a grandchild G in the chain, then G has sequence number 3; and so on.

Synapse ensures that each persisted event belongs to exactly one chain, and tracks how the chains are connected to one another. This allows us to efficiently answer reachability queries. Doing so uses less storage than tracking reachability on an event-by-event basis, particularly when we have fewer and longer chains. See

Jagadish, H. (1990). [A compression technique to materialize transitive closure](#). *ACM Transactions on Database Systems (TODS)*, 15*(4)*, 558-598.

for the original idea or

Y. Chen, Y. Chen, [An efficient algorithm for answering graph reachability queries](#), in: 2008 IEEE 24th International Conference on Data Engineering, April 2008, pp. 893–902. (PDF available via [Google Scholar](#).)

for a more modern take.

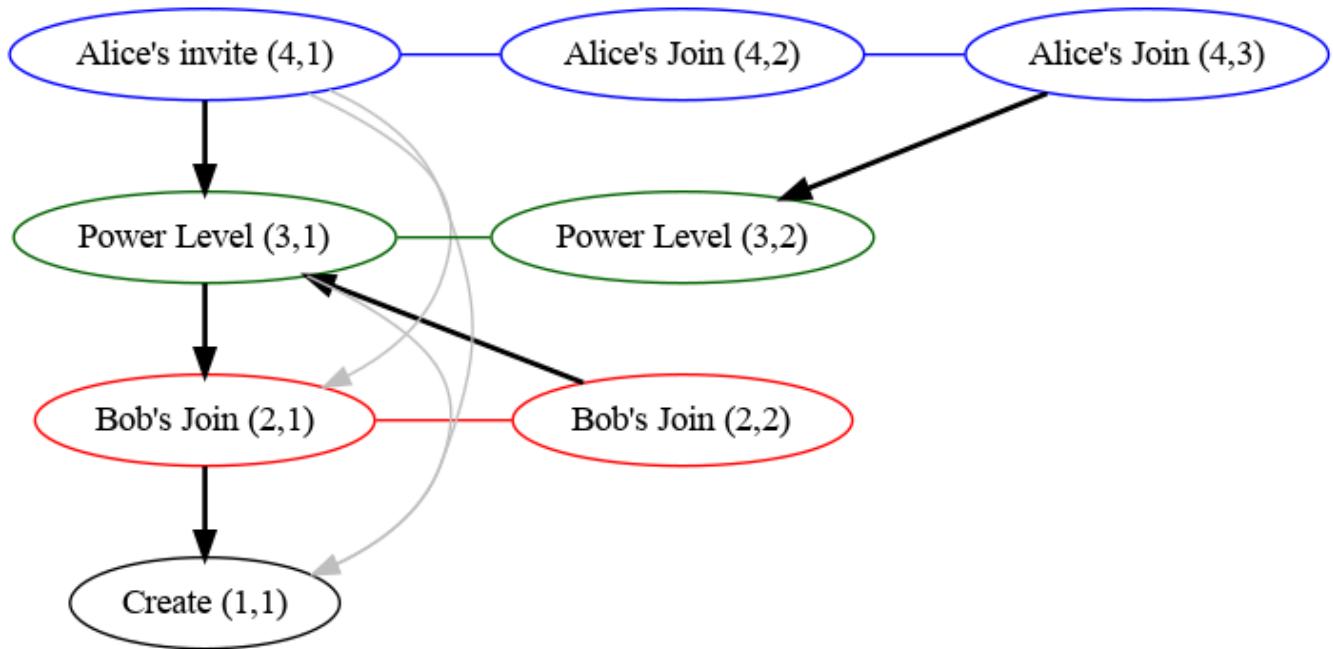
In practical terms, the chain cover assigns every event a *chain ID* and *sequence number* (e.g. $(5,3)$), and maintains a map of *links* between events in chains (e.g. $(5,3) \rightarrow (2,4)$) such that A is reachable by B (i.e. A is in the auth chain of B) if and only if either:

1. A and B have the same chain ID and A 's sequence number is less than B 's sequence number; or
2. there is a link L between B 's chain ID and A 's chain ID such that $L.start_seq_no \leq B.seq_no$ and $A.seq_no \leq L.end_seq_no$.

There are actually two potential implementations, one where we store links from each chain to every other reachable chain (the transitive closure of the links graph), and one where we remove redundant links (the transitive reduction of the links graph) e.g. if we have chains $c3 \rightarrow c2 \rightarrow c1$ then the link $c3 \rightarrow c1$ would not be stored. Synapse uses the former implementation so that it doesn't need to recurse to test reachability between chains. This trades-off extra storage in order to save CPU cycles and DB queries.

Example

An example auth graph would look like the following, where chains have been formed based on type/state_key and are denoted by colour and are labelled with **(chain ID, sequence number)**. Links are denoted by the arrows (links in grey are those that would be removed in the second implementation described above).



Note that we don't include all links between events and their auth events, as most of those links would be redundant. For example, all events point to the create event, but each chain only needs the one link from its base to the create event.

Using the Index

This index can be used to calculate the auth chain difference of the state sets by looking at the chain ID and sequence numbers reachable from each state set:

1. For every state set lookup the chain ID/sequence numbers of each state event
2. Use the index to find all chains and the maximum sequence number reachable from each state set.
3. The auth chain difference is then all events in each chain that have sequence numbers between the maximum sequence number reachable from *any* state set and the minimum reachable by *all* state sets (if any).

Note that steps 2 is effectively calculating the auth chain for each state set (in terms of chain IDs and sequence numbers), and step 3 is calculating the difference between the union and intersection of the auth chains.

Worked Example

For example, given the above graph, we can calculate the difference between state sets consisting of:

1. s_1 : Alice's invite $(4,1)$ and Bob's second join $(2,2)$; and
2. s_2 : Alice's second join $(4,3)$ and Bob's first join $(2,1)$.

Using the index we see that the following auth chains are reachable from each state set:

1. s_1 : $(1,1)$, $(2,2)$, $(3,1)$ & $(4,1)$
2. s_2 : $(1,1)$, $(2,1)$, $(3,2)$ & $(4,3)$

And so, for each the ranges that are in the auth chain difference:

1. Chain 1: None, (since everything can reach the create event).
2. Chain 2: The range $(1, 2]$ (i.e. just 2), as 1 is reachable by all state sets and the maximum reachable is 2 (corresponding to Bob's second join).
3. Chain 3: Similarly the range $(1, 2]$ (corresponding to the second power level).
4. Chain 4: The range $(1, 3]$ (corresponding to both of Alice's joins).

So the final result is: Bob's second join $(2,2)$, the second power level $(3,2)$ and both of Alice's joins $(4,2)$ & $(4,3)$.

Media Repository

Synapse implementation-specific details for the media repository

The media repository

- stores avatars, attachments and their thumbnails for media uploaded by local users.
- caches avatars, attachments and their thumbnails for media uploaded by remote users.
- caches resources and thumbnails used for URL previews.

All media in Matrix can be identified by a unique [MXC URI](#), consisting of a server name and media ID:

```
mx: //<server-name>/<media-id>
```

Local Media

Synapse generates 24 character media IDs for content uploaded by local users. These media IDs consist of upper and lowercase letters and are case-sensitive. Other homeserver implementations may generate media IDs differently.

Local media is recorded in the `local_media_repository` table, which includes metadata such as MIME types, upload times and file sizes. Note that this table is shared by the URL cache, which has a different media ID scheme.

Paths

A file with media ID `aabbcccccccccccccccccc` and its `128x96` `image/jpeg` thumbnail, created by scaling, would be stored at:

```
local_content/aa/bb/cccccccccccccccccc  
local_thumbnails/aa/bb/cccccccccccccccccc/128-96-image-jpeg-scale
```

Remote Media

When media from a remote homeserver is requested from Synapse, it is assigned a local `filesystem_id`, with the same format as locally-generated media IDs, as described above.

A record of remote media is stored in the `remote_media_cache` table, which can be used to map remote MXC URLs (server names and media IDs) to local `filesystem_ids`.

Paths

A file from `matrix.org` with `filesystem_id` `aabbcccccccccccccccccccc` and its `128x96 image/jpeg` thumbnail, created by scaling, would be stored at:

```
remote_content/matrix.org/aa/bb/cccccccccccccccccccc  
remote_thumbnail/matrix.org/aa/bb/cccccccccccccccccccc/128-96-image-jpeg-scale
```

Older thumbnails may omit the thumbnailing method:

```
remote_thumbnail/matrix.org/aa/bb/cccccccccccccccccccc/128-96-image-jpeg
```

Note that `remote_thumbnail/` does not have an `s`.

URL Previews

When generating previews for URLs, Synapse may download and cache various resources, including images. These resources are assigned temporary media IDs of the form `yyyy-mm-dd_aaaaaaaaaaaaaaaa`, where `yyyy-mm-dd` is the current date and `aaaaaaaaaaaaaaa` is a random sequence of 16 case-sensitive letters.

The metadata for these cached resources is stored in the `local_media_repository` and `local_media_repository_url_cache` tables.

Resources for URL previews are deleted after a few days.

Paths

The file with media ID `yyyy-mm-dd_aaaaaaaaaaaaaaaa` and its `128x96 image/jpeg` thumbnail, created by scaling, would be stored at:

```
url_cache/yyyy-mm-ddaaaaaaaaaaaaaaa  
url_cache_thumbnails/yyyy-mm-ddaaaaaaaaaaaaaaa/128-96-image-jpeg-scale
```

Room and User Statistics

Synapse maintains room and user statistics in various tables. These can be used for administrative purposes but are also used when generating the public room directory.

Synapse Developer Documentation

High-Level Concepts

Definitions

- **subject**: Something we are tracking stats about – currently a room or user.
- **current row**: An entry for a subject in the appropriate current statistics table. Each subject can have only one.

Overview

Stats correspond to the present values. Current rows contain the most up-to-date statistics for a room. Each subject can only have one entry.

Deprecation Policy for Platform Dependencies

Synapse has a number of platform dependencies, including Python, Rust, PostgreSQL and SQLite. This document outlines the policy towards which versions we support, and when we drop support for versions in the future.

Policy

Synapse follows the upstream support life cycles for Python and PostgreSQL, i.e. when a version reaches End of Life Synapse will withdraw support for that version in future releases.

Details on the upstream support life cycles for Python and PostgreSQL are documented at <https://endoflife.date/python> and <https://endoflife.date/postgresql>.

A Rust compiler is required to build Synapse from source. For any given release the minimum required version may be bumped up to a recent Rust version, and so people building from source should ensure they can fetch recent versions of Rust (e.g. by using [rustup](#)).

The oldest supported version of SQLite is the version provided by [Debian oldstable](#).

Context

It is important for system admins to have a clear understanding of the platform requirements of Synapse and its deprecation policies so that they can effectively plan upgrading their infrastructure ahead of time. This is especially important in contexts where upgrading the infrastructure requires auditing and approval from a security team, or where otherwise upgrading is a long process.

By following the upstream support life cycles Synapse can ensure that its dependencies continue to get security patches, while not requiring system admins to constantly update their platform dependencies to the latest versions.

For Rust, the situation is a bit different given that a) the Rust foundation does not generally support older Rust versions, and b) the library ecosystem generally bump their minimum support Rust versions frequently. In general, the Synapse team will try to avoid updating the dependency on Rust to the absolute latest version, but introducing a formal policy is hard given the constraints of the ecosystem.

On a similar note, SQLite does not generally have a concept of "supported release"; bugfixes are published for the latest minor release only. We chose to track Debian's oldstable as this is relatively conservative, predictably updated and is consistent with the `.deb` packages released by Matrix.org.

Summary of performance impact of running on resource constrained devices such as SBCs

I've been running my homeserver on a cubietruck at home now for some time and am often replying to statements like "you need loads of ram to join large rooms" with "it works fine for me". I thought it might be useful to curate a summary of the issues you're likely to run into to help as a scaling-down guide, maybe highlight these for development work or end up as documentation. It seems that once you get up to about 4x1.5GHz arm64 4GiB these issues are no longer a problem.

- **Platform:** 2x1GHz armhf 2GiB ram [Single-board computers](#), SSD, postgres.

Presence

This is the main reason people have a poor matrix experience on resource constrained homeservers. Element web will frequently be saying the server is offline while the python process will be pegged at 100% cpu. This feature is used to tell when other users are active (have a client app in the foreground) and therefore more likely to respond, but requires a lot of network activity to maintain even when nobody is talking in a room.



Connectivity to the server has been lost.

Sent messages will be stored until your connection has returned.

While synapse does have some performance issues with presence [#3971](#), the fundamental problem is that this is an easy feature to implement for a centralised service at nearly no overhead, but federation makes it combinatorial [#8055](#). There is also a client-side config option which disables the UI and idle tracking [enable_presence_by_hs_url](#) to blacklist the largest instances but I didn't notice much difference, so I recommend disabling the feature entirely at the server level as well.

Joining

Joining a "large", federated room will initially fail with the below message in Element web, but waiting a while (10-60mins) and trying again will succeed without any issue. What counts as "large" is not message history, user count, connections to homeservers or even a simple count of the state events, it is instead how long the state resolution algorithm takes. However, each of those numbers are reasonable proxies, so we can use them as estimates since user count is one of the few things you see before joining.



Failed to join room

There was an error joining the room

OK

This is [#1211](#) and will also hopefully be mitigated by peeking [matrix-org/matrix-doc#2753](#) so at least you don't need to wait for a join to complete before finding out if it's the kind of room you want. Note that you should first disable presence, otherwise it'll just make the situation worse [#3120](#). There is a lot of database interaction too, so make sure you've [migrated your data](#) from the default sqlite to postgresql. Personally, I recommend patience - once the initial join is complete there's rarely any issues with actually interacting with the room, but if you like you can just block "large" rooms entirely.

Sessions

Anything that requires modifying the device list [#7721](#) will take a while to propagate, again taking the client "Offline" until it's complete. This includes signing in and out, editing the public name and verifying e2ee. The main mitigation I recommend is to keep long-running sessions open e.g. by using Firefox SSB "Use this site in App mode" or Chromium PWA "Install Element".

Recommended configuration

Put the below in a new file at `/etc/matrix-synapse/conf.d/sbc.yaml` to override the defaults in `homeserver.yaml`.

```

# Disable presence tracking, which is currently fairly resource intensive
# More info: https://github.com/matrix-org/synapse/issues/9478
use_presence: false

# Set a small complexity limit, preventing users from joining large rooms
# which may be resource-intensive to remain a part of.
#
# Note that this will not prevent users from joining smaller rooms that
# eventually become complex.
limit_remote_rooms:
  enabled: true
  complexity: 3.0

# Database configuration
database:
  # Use postgres for the best performance
  name: psycopg2
  args:
    user: matrix-synapse
    # Generate a long, secure password using a password manager
    password: hunter2
    database: matrix-synapse
    host: localhost

```

Currently the complexity is measured by [current_state_events / 500](#). You can find join times and your most complex rooms like this:

```

admin@homeserver:~$ zgrep '/client/r0/join/' /var/log/matrix-
synapse/homeserver.log* | awk '{print $18, $25}' | sort --human-numeric-sort
29.922sec/-0.002sec /_matrix/client/r0/join/%23debian-fasttrack%3Apoddery.com
182.088sec/0.003sec /_matrix/client/r0/join/%23decentralizedweb-
general%3Amatrix.org
911.625sec/-570.847sec /_matrix/client/r0/join/%23synapse%3Amatrix.org

admin@homeserver:~$ sudo --user postgres psql matrix-synapse --command 'select
canonical_alias, joined_members, current_state_events from room_stats_state
natural join room_stats_current where canonical_alias is not null order by
current_state_events desc fetch first 5 rows only'
  canonical_alias | joined_members | current_state_events
-----+-----+-----+
#_oftc_#debian:matrix.org | 871 | 52355
#matrix:matrix.org | 6379 | 10684
#irc:matrix.org | 461 | 3751
#decentralizedweb-general:matrix.org | 997 | 1509
#whatsapp:maunium.net | 554 | 854

```