Matrix Specification

Matrix defines a set of open APIs for decentralised communication, suitable for securely publishing, persisting and subscribing to data over a global open federation of servers with no single point of control. Uses include Instant Messaging (IM), Voice over IP (VoIP) signalling, Internet of Things (IoT) communication, and bridging together existing communication silos - providing the basis of a new open real-time communication ecosystem.

To propose a change to the Matrix Spec, see the explanations at Proposals for Spec Changes to Matrix.

Table of Contents

The specification consists of the following parts:

API	Version	Description
Client-Server API	`r0.4.0`	Interaction between clients and servers
Server-Server API	`r0.1.1`	Federation between servers
Application Service API	`r0.1.0`	Privileged server plugins
Identity Service API	`r0.1.0`	Mapping of third party IDs to Matrix IDs
Push Gateway API	`r0.1.0`	Push notifications for Matrix events

Additionally, this introduction page contains the key baseline information required to understand the specific APIs, including the sections on room versions and overall architecture.

The Appendices contain supplemental information not specific to one of the above APIs.

The Matrix Client-Server API Swagger Viewer is useful for browsing the Client-Server API.

2 Introduction to the Matrix APIs

Warning: The Matrix specification is still evolving: the APIs are not yet frozen and this document is in places a work in progress or stale. We have made every effort to clearly flag areas which are still being finalised. We're publishing it at this point because it's complete enough to be more than useful and provide a canonical reference to how Matrix is evolving. Our end goal is to mirror WHATWG's Living Standard.

Matrix is a set of open APIs for open-federated Instant Messaging (IM), Voice over IP (VoIP) and Internet of Things (IoT) communication, designed to create and support a new global real-time communication ecosystem. The intention is to provide an open decentralised pubsub layer for the internet for securely persisting and publishing/subscribing JSON objects. This specification is the ongoing result of standardising the APIs used by the various components of the Matrix ecosystem to communicate with one another.

The principles that Matrix attempts to follow are:

- Pragmatic Web-friendly APIs (i.e. JSON over REST)

- - Fully open standard - publicly documented standard with no IP or patent licensing encumbrances
  - Fully open source reference implementation - liberally-licensed example implementations with no IP or patent licensing encumbrances
- Empowering the end-user
  - The user should be able to choose the server and clients they use
  - The user should be control how private their communication is
  - The user should know precisely where their data is stored
- Fully decentralised - no single points of control over conversations or the network as a whole
- Learning from history to avoid repeating it
  - Trying to take the best aspects of XMPP, SIP, IRC, SMTP, IMAP and NNTP whilst trying to avoid their failings

The functionality that Matrix provides includes:

- Creation and management of fully distributed chat rooms with no single points of control or failure
- Eventually-consistent cryptographically secure synchronisation of room state across a global open network of federated servers and services
- Sending and receiving extensible messages in a room with (optional) end-to-end encryption
- Extensible user management (inviting, joining, leaving, kicking, banning) mediated by a power-level based user privilege system.
- Extensible room state management (room naming, aliasing, topics, bans)
- Extensible user profile management (avatars, display names, etc)
- Managing user accounts (registration, login, logout)
- Use of 3rd Party IDs (3PIDs) such as email addresses, phone numbers, Facebook accounts to authenticate, identify and discover users on Matrix.
- Trusted federation of identity servers for:
  - Publishing user public keys for PKI
  - Mapping of 3PIDs to Matrix IDs

The end goal of Matrix is to be a ubiquitous messaging layer for synchronising arbitrary data between sets of people, devices and services - be that for instant messages, VoIP call setups, or any other objects that need to be reliably and persistently pushed from A to B in an interoperable and federated manner.

2.1 Spec Change Proposals

3 Architecture

Matrix defines APIs for synchronising extensible JSON objects known as "events" between compatible clients, servers and services. Clients are typically messaging/VoIP applications or IoT devices/hubs and communicate by synchronising communication history with their "homeserver" using the "Client-Server API". Each homeserver stores the communication history and account information for all of its clients, and shares data with the wider Matrix ecosystem by synchronising communication history with other homeservers and their clients.

Clients typically communicate with each other by emitting events in the context of a virtual "room". Room data is replicated across *all of the homeservers* whose users are participating in a given room. As such, *no single homeserver has control or ownership over a given room*. Homeservers model communication history as a partially ordered graph of events known as the room's "event graph", which is synchronised with eventual consistency between the participating servers using the "Server-Server API". This process of synchronising shared conversation history between homeservers run b different parties is called "Federation". Matrix optimises for the Availability and Partitioned properties of CAP theorem at the expense of Consistency.

For example, for client A to send a message to client B, client A performs an HTTP PUT of the required JSON event on its homeserver (HS) using the client-server API. A's HS appends this event to its copy of the room's event graph, signing the message in the context of the graph for integrity. A's HS then replicates the message to B's HS by performing an HTTP PUT using the server-server API. B's HS authenticates the request, validates the event's signature, authorises the event's contents and then adds it to its copy of the room's event graph. Client B then receives the message from his homeserver via a long-lived GET request.

```
                  How data flows between clients
                  ==============================

{ Matrix client A }                             { Matrix client B }
    ^          |                                    ^          |
    |  events  |  Client-Server API                 |  events  |
    |          V                                    |          V
+------------------+                            +------------------+
|                  |---------( HTTPS )--------->|                  |
|   homeserver     |                            |   homeserver     |
|                  |<---------( HTTPS )----------|                  |
+------------------+      Server-Server API     +------------------+
                       History Synchronisation
                           (Federation)
```

```
@localpart:domain
```

See 'Identifier Grammar' the appendices for full details of the structure of user IDs.

3.2 Devices

The Matrix specification has a particular meaning for the term "device". As a user, I might have several devices: a desktop client, some web browsers, ar Android device, an iPhone, etc. They broadly relate to a real device in the physical world, but you might have several browsers on a physical device, or several Matrix client applications on a mobile device, each of which would be its own device.

Devices are used primarily to manage the keys used for end-to-end encryption (each device gets its own copy of the decryption keys), but they also hel users manage their access - for instance, by revoking access to particular devices.

When a user first uses a client, it registers itself as a new device. The longevity of devices might depend on the type of client. A web client will probably drop all of its state on logout, and create a new device every time you log in, to ensure that cryptography keys are not leaked to a new user. In a mobile client, it might be acceptable to reuse the device if a login session expires, provided the user is the same.

Devices are identified by a `device_id`, which is unique within the scope of a given user.

A user may assign a human-readable display name to a device, to help them manage their devices.

3.3 Events

All data exchanged over Matrix is expressed as an "event". Typically each client action (e.g. sending a message) correlates with exactly one event. Each event has a `type` which is used to differentiate different kinds of data. `type` values MUST be uniquely globally namespaced following Java's package naming conventions, e.g. `com.example.myapp.event`. The special top-level namespace `m.` is reserved for events defined in the Matrix specification - for instance `m.room.message` is the event type for instant messages. Events are usually sent in the context of a "Room".

Events exchanged in the context of a room are stored in a directed acyclic graph (DAG) called an "event graph". The partial ordering of this graph gives the chronological ordering of events within the room. Each event in the graph has a list of zero or more "parent" events, which refer to any preceding events which have no chronological successor from the perspective of the homeserver which created the event.

Typically an event has a single parent: the most recent message in the room at the point it was sent. However, homeservers may legitimately race with each other when sending messages, resulting in a single event having multiple successors. The next event added to the graph thus will have multiple parents. Every event graph has a single root event with no parent.

To order and ease chronological comparison between the events within the graph, homeservers maintain a `depth` metadata field on each event. An event's `depth` is a positive integer that is strictly greater than the depths of any of its parents. The root event should have a depth of 1. Thus if one even is before another, then it must have a strictly smaller depth.

3.5 Room structure

A room is a conceptual place where users can send and receive events. Events are sent to a room, and all participants in that room with sufficient access will receive the event. Rooms are uniquely identified internally via "Room IDs", which have the form:

```
!opaque_id:domain
```

There is exactly one room ID for each room. Whilst the room ID does contain a domain, it is simply for globally namespacing room IDs. The room does NOT reside on the domain specified.

See 'Identifier Grammar' in the appendices for full details of the structure of a room ID.

The following conceptual diagram shows an `m.room.message` event being sent to the room `!qporfwt:matrix.org`:

```
 { @alice:matrix.org }                             { @bob:example.org }
         |                                                 ^
         |                                                 |
[HTTP POST]                                   [HTTP GET]
Room ID: !qporfwt:matrix.org                  Room ID: !qporfwt:matrix.org
Event type: m.room.message                    Event type: m.room.message
```

```
|   matrix.org     |                           |   example.org    |
+------------------+                           +------------------+
        |                                                ^
        |         [HTTP PUT]                             |
        |         Room ID: !qporfwt:matrix.org           |
        |         Event type: m.room.message             |
        |         Content: { JSON object }               |
        `-------> Pointer to the preceding message  ------`
                  PKI signature from matrix.org
                  Transaction-layer metadata
                  PKI Authorization header

               ...................................
              |           Shared Data             |
              | State:                            |
              |   Room ID: !qporfwt:matrix.org    |
              |   Servers: matrix.org, example.org |
              |   Members:                        |
              |    - @alice:matrix.org            |
              |    - @bob:example.org             |
              | Messages:                         |
              |   - @alice:matrix.org             |
              |     Content: { JSON object }      |
              |...................................|
```

Federation maintains *shared data structures* per-room between multiple homeservers. The data is split into `message events` and `state events`.

Message events:

These describe transient 'once-off' activity in a room such as an instant messages, VoIP call setups, file transfers, etc. They generally describe communication activity.

State events:

These describe updates to a given piece of persistent information ('state') related to a room, such as the room's name, topic, membership, participating servers, etc. State is modelled as a lookup table of key/value pairs per room, with each key being a tuple of `state_key` and `event type`. Each state event updates the value of a given key.

The state of the room at a given point is calculated by considering all events preceding and including a given event in the graph. Where events describe the same state, a merge conflict algorithm is applied. The state resolution algorithm is transitive and does not depend on server state, as it must consistently select the same event irrespective of the server or the order the events were received in. Events are signed by the originating server (the

3.5.1 Room Aliases

Each room can also have multiple "Room Aliases", which look like:

```
#room_alias:domain
```

See 'Identifier Grammar' in the appendices for full details of the structure of a room alias.

A room alias "points" to a room ID and is the human-readable label by which rooms are publicised and discovered. The room ID the alias is pointing to can be obtained by visiting the domain specified. Note that the mapping from a room alias to a room ID is not fixed, and may change over time to point to a different room ID. For this reason, Clients SHOULD resolve the room alias to a room ID once and then use that ID on subsequent requests.

When resolving a room alias the server will also respond with a list of servers that are in the room that can be used to join via.

```
     HTTP GET
#matrix:example.org      !aaabaa:matrix.org
        |                    ^
        |                    |
 _______V____________________|____
|          example.org            |
| Mappings:                       |
| #matrix >> !aaabaa:matrix.org   |
| #golf   >> !wfeiofh:sport.com   |
| #bike   >> !4rguxf:matrix.org   |
|_________________________________|
```

3.6 Identity

Users in Matrix are identified via their Matrix user ID. However, existing 3rd party ID namespaces can also be used in order to identify Matrix users. A Matrix "Identity" describes both the user ID and any other existing IDs from third party namespaces *linked* to their account. Matrix users can *link* third-party IDs (3PIDs) such as email addresses, social network accounts and phone numbers to their user ID. Linking 3PIDs creates a mapping from a 3PID t

Usage of an IS is not required in order for a client application to be part of the Matrix ecosystem. However, without one clients will not be able to look up user IDs using 3PIDs.

3.7 Profiles

Users may publish arbitrary key/value data associated with their account - such as a human readable display name, a profile photo URL, contact information (email address, phone numbers, website URLs etc).

3.8 Private User Data

Users may also store arbitrary private key/value data in their account - such as client preferences, or server configuration settings which lack any other dedicated API. The API is symmetrical to managing Profile data.

4 Room Versions

Rooms are central to how Matrix operates, and have strict rules for what is allowed to be contained within them. Rooms can also have various algorithms that handle different tasks, such as what to do when two or more events collide in the underlying DAG. To allow rooms to be improved upor through new algorithms or rules, "room versions" are employed to manage a set of expectations for each room. New room versions are assigned as needed.

There is no implicit ordering or hierarchy to room versions, and their principles are immutable once placed in the specification. Although there is a recommended set of versions, some rooms may benefit from features introduced by other versions. Rooms move between different versions by "upgrading" to the desired version. Due to versions not being ordered or hierarchical, this means a room can "upgrade" from version 2 to version 1, if it is so desired.

4.1 Room version grammar

contain only the characters `a-z`, `0-9`, `.`, and `-`.

Room versions are not intended to be parsed and should be treated as opaque identifiers. Room versions consisting only of the characters `0-9` and `.` are reserved for future versions of the Matrix protocol.

The complete grammar for a legal room version is:

```
room_version = 1*room_version_char
room_version_char = DIGIT
                  / %x61-7A         ; a-z
                  / "-" / "."
```

Examples of valid room versions are:

- `1` (would be reserved by the Matrix protocol)
- `1.2` (would be reserved by the Matrix protocol)
- `1.2-beta`
- `com.example.version`

4.2 Complete list of room versions

Room versions are divided into two distinct groups: stable and unstable. Stable room versions may be used by rooms safely. Unstable room versions are everything else which is either not listed in the specification or flagged as unstable for some other reason. Versions can switch between stable and unstable periodically for a variety of reasons, including discovered security vulnerabilities and age.

Clients should not ask room administrators to upgrade their rooms if the room is running a stable version. Servers SHOULD use room version 1 as the default room version when creating new rooms.

The available room versions are:

- Version 1 - **Stable**. The current version of most rooms.

5 Specification Versions

The specification for each API is versioned in the form `rX.Y.Z`.

- A change to `X` reflects a breaking change: a client implemented against `r1.0.0` may need changes to work with a server which supports (only) `r2.0.0`.
- A change to `Y` represents a change which is backwards-compatible for existing clients, but not necessarily existing servers: a client implemented against `r1.1.0` will work without changes against a server which supports `r1.2.0`; but a client which requires `r1.2.0` may not work correctly with a server which implements only `r1.1.0`.
- A change to `Z` represents a change which is backwards-compatible on both sides. Typically this implies a clarification to the specification, rather than a change which must be implemented.

6 License

The Matrix specification is licensed under the Apache License, Version 2.0.