Application services | Matrix.org

Originalartikel

Backup

<html> <p>Application services are distinct modules which which sit alongside a homeserver providing arbitrary extensible functionality decoupled from the homeserver implementation. Just like the rest of Matrix, they communicate via HTTP using JSON. Application services function in a very similar way to traditional clients, but they are given much more power than a normal client. They can reserve entire namespaces of room aliases and user IDs for their own purposes. They can silently monitor events in rooms, or any events directed at any user ID. This power allows application services to have extremely useful abilities which overall enhance the end user experience.</p> <p>One of the main use cases for application services is protocol bridges. Our Matrix server on Matrix.org links in to various IRC channels and networks. This functionality was initially implemented as a simple bot which resided as a user on the Matrix rooms we wanted to link to freenode channels (#matrix, #matrix-dev, #openwebrtc and #vuc etc). There was nothing special about this bot; it is just treated as a client. However, as we started to rely on it more and more though, we realised that there were things that were impossible for simple client-side bots to do by themselves - for example, the bot could not reserve the virtual user IDs it wanted to create, and could not lazily bridge arbitrary IRC rooms on-the-fly - and this spurred the development of Application Services.</p> <h3 id=„some-of-the-features-of-the-irc-application-service-we-have-since-implemented-include“>Some of the features of the IRC application service we have since implemented include:</h3> <ul><li>Specific channel-to-matrix room bridging : This is what the original IRC bot did. You can specify specific channels and specific room IDs, and messages will be bridged.</li>

```
<li>Dynamic channel-to-matrix room bridging : This allows Matrix users to
join any channel on an IRC network, rather than being forced to use one of
the specific channels configured.</li>
<li>Two-way PM support : IRC users can PM the virtual &#8220;M-&#8220; users
and private Matrix rooms will be created. Likewise, Matrix users can invite
the virtual &#8220;@irc_Nick:domain&#8221; user IDs to a room and a PM to
the IRC nick will be made.</li>
<li>IRC nick changing support: Matrix users are no longer forced to use
&#8220;M-&#8220; nicks and can change them by sending &#8220;!nick&#8221;
messages directly to the bridge.</li>
<li>Ident support: This allows usernames to be authenticated for virtual IRC
clients, which means IRC bans can be targeted at the Matrix user rather than
the entire application service.</li>
```

</ul><h3 id=„the-use-of-the-application-services-api-means“>The use of the Application Services API means:</h3> <ul><li>The bot can reserve user IDs. This prevents humans from registering for @irc_&#8230; user IDs which would then clash with the operation of the bot.</li>

```
<li>The bot can reserve room aliases. This prevents humans from register for
#irc_&#8230; aliases which would then clash with the operation of the
bot.</li>
<li>The bot can trivially manage hundreds of users. Events are pushed to the
application service directly. If you tried to do this as a client-side bot,
you would need one event stream connection per virtual user.</li>
```

```
<li>The bot can lazily create rooms on demand. This means Matrix users can
join non-existent room aliases and have the application service quickly
track an IRC channel and create a room with that alias, allowing the join
request to succeed.</li>
```

</ul><h3 id=„implementation-details“>Implementation details:</h3> <ul><li>Written in Node.js, designed to be run using

```
forever
```

.</li>

```
<li>Built on the generic <a
href="http://github.com/matrix-org/matrix-appservice-node">matrix-appservice
-node</a> framework.</li>
<li>Supports sending metrics in statsd format.</li>
<li>Uses matrix-appservice-node to provide a standardised interface when
writing application services, rather than an explicit web framework (though
under the hood matrix-appservice-node is using Express).</li>
```

</ul><p>At present, the IRC application service is in beta, and is being run on #matrix and #matrix-dev. If you want to give it a go, <a title=„Matrix-IRC Application Service“ href=„https://github.com/matrix-org/matrix-appservice-irc“>check it out on Github</a>!</p> <p>Application services have enormous potential for creating new and exciting ways to transform and enhance the core Matrix protocol. For example, you could aggregate information from multiple rooms into a summary room, or create throwaway virtual user accounts to proxy messages for a fixed user ID on-the-fly. As you may expect, all of this power assumes a high degree of trust between application services and homeservers. Only homeserver admins can allow an application service to link up with their homeserver, and the application service is in no way federated to other homeservers. You can think of application services as additional logic on the homeserver itself, without messing around with the book-keeping that homeservers have to do. This makes adding useful functionality very easy.</p> <h3 id=„example“>Example</h3> <p>The application service (AS) API itself uses webhooks to communicate from the homeserver to the AS:</p> <ul><li>Room Alias Query API : The homeserver hits a URL on your application server to see if a room alias exists.</li>

```
<li>User Query API : The homeserver hits a URL on your application server to
see if a user ID exists.</li>
<li>Push API : The homeserver hits a URL on your application server to
notify you of new events for your users and rooms.</li>
```

</ul><p>A very basic application service may want to log all messages in rooms which have an alias starting with &#8220;#logged_&#8221; (side note: logging won&#8217;t work if these rooms are using end-to-end encryption).</p> <p>Here&#8217;s an example of a very basic application service using Python (with Flask and Requests) which logs room activity:</p> <pre> # app_service.py:

```
  import json, requests&#160; # we will use this later
  from flask import Flask, jsonify, request
  app = Flask(__name__)
  @app.route("/transactions/&amp;lt;transaction&amp;gt;", methods=["PUT"])
  def on_receive_events(transaction):
      events = request.get_json()["events"]
      for event in events:
          print "User: %s Room: %s" % (event["user_id"], event["room_id"])
          print "Event Type: %s" % event["type"]
          print "Content: %s" % event["content"]
      return jsonify({})
  if __name__ == "__main__":
      app.run()
```

</pre> <p>Set your new application service running on port 5000&#160;with:</p> <pre> python app_service.py </pre> <p>The homeserver needs to know that the application service exists before it will send requests to it. This is done via a registration YAML file which is specified in Synapse&#8217;s main config file e.g.

```
homeserver.yaml
```

. The server admin needs to add the application service registration configuration file as an entry to this file.</p> <pre> # homeserver.yaml

```
  app_service_config_files:
    - "/path/to/appservice/registration.yaml"
```

</pre> <p>NB: Note the &#8220;-&#8220; at the start; this indicates a list element. The registration file

```
registration.yaml
```

should look like:</p> <pre> # registration.yaml

# An ID which is unique across all application services on your homeserver. This should never be changed once set. id: „something-good“

```
  # this is the base URL of the application service
  url: "http://localhost:5000"
  # This is the token that the AS should use as its access_token when using
the Client-Server API
  # This can be anything you want.
  as_token: wfghWEGh3wgWHEf3478sHFWE
  # This is the token that the HS will use when sending requests to the AS.
  # This can be anything you want.
  hs_token: ugw8243igya57aaABGFfgeyu
  # this is the local part of the desired user ID for this AS (in this case
@logging:localhost)
  sender_localpart: logging
  namespaces:
    users: []
    rooms: []
    aliases:
      - exclusive: false
```

```
        regex: "#logged_.*"
```

</pre> <p><strong>You will need to restart the homeserver after editing the config file before it will take effect.</strong></p> <p>To test everything is working correctly, go ahead and explicitly create a room with the alias &#8220;#logged_test:localhost&#8221; and send a message into the room: the HS will relay the message to the AS by PUTing to /transactions/&lt;tid&gt; and you should see your AS print the event on the terminal. This will monitor any room which has an alias prefix of &#8220;#logged_&#8221;, but it won&#8217;t lazily create room aliases if they don&#8217;t already exist. This means it will only&#160;log messages in the room you created before: #logged_test:localhost. Try joining the room &#8220;#logged_test2:localhost&#8221; without creating it, and it will fail. Let&#8217;s fix that and add in lazy room creation:</p> <pre> @app.route(„/rooms/&amp;lt;alias&amp;gt;“)

```
  def query_alias(alias):
      alias_localpart = alias.split(":")[0][1:]
      requests.post(
          # NB: "TOKEN" is the as_token referred to in registration.yaml
"http://localhost:8008/_matrix/client/api/v1/createRoom?access_token=TOKEN",
          json.dumps({
              "room_alias_name": alias_localpart
          }),
          headers={"Content-Type":"application/json"}
      )
      return jsonify({})
```

</pre> <p>This makes the application service lazily create a room with the requested alias whenever the HS queries the AS for the existence of that alias (when users try to join that room), allowing any room with the alias prefix #logged_ to be sent to the AS. Now try joining the room &#8220;#logged_test2:localhost&#8221; and it will work as you&#8217;d expect. &#160;You can see that if this were a real bridge, the AS would have checked for the existence of #logged_test2 in the remote network, and then lazily-created it in Matrix as required.</p> <p>Application services are powerful components which extend the functionality of homeservers, but they are limited. They can only ever function in a &#8220;passive&#8221; way.&#160;For example, you cannot implement an application service which censors swear words in rooms, because there is no way to prevent the event from being sent. Aside from the&#160;fact that censoring will not work when using end-to-end encryption, all federated homeservers would also need to reject the event in order to stop developing an&#160;inconsistent event graph. To &#8220;actively&#8221; monitor events, another component called a &#8220;Policy Server&#8221; is required, which is beyond the scope of this post. &#160;Also, Application Services can result in a performance bottleneck, as all events on the homeserver must be ordered and sent to the registered application services. &#160;If you are bridging huge amounts of traffic, you may be better off having your bridge directly talk the Server-Server federation API&#160;rather than the simpler Application Service API.</p> <p>I hope this demonstrates how easy it is to create an application service, along with a few ideas of the kinds of things you can do with them. Obvious uses include build protocol bridges, search engines, invisible bots, etc. For more information on the AS HTTP API, check out the new <a href=„http://matrix.org/docs/spec/#application-service-api“>Application Service API</a> section in the spec, or the raw drafts and spec in <a href=„https://github.com/matrix-org/matrix-doc/“ target=„_blank“>https://github.com/matrix-org/matrix-doc/</a>.</p> </html>