Advanced Synapse setup with Let’s Encrypt

Originalartikel

Backup

<html>

```
                <p>So, you&#8217;ve installed an configured synapse and
started chatting from your very own Matrix home server? What&#8217;s the
next step? Well, right now you&#8217;re probably accessing your new
home server over plaintext HTTP, which is bad, particularly because
you&#8217;ll be sending your password over this connection when you log in.
You could connect to Synapse&#8217;s secure HTTP port, but your browser
won&#8217;t trust it by default because you&#8217;d normally need to pay for
a certificate that your browser would recognise. That is, until
recently!</p>
```

Let’s Encrypt is a new initiative that issues SSL certificates free of charge, in an effort to make SSL universal on the Internet. In this blog post, we’ll be walking through an example of how we can use this service to get ourselves a securely hosted Synapse. We’re going to assume you have a Synapse installed and listening on the standard ports, 8008 and 8448. If not, follow the Synapse README and come back here when you’re done. Everybody ready? Okay, good. So, in order to get a certificate from Let’s Encrypt, we need to prove that we own our domain. The simplest way to do this is to host some special files on our web server. Now, Synapse won’t do this. We could use a separate web server, but then we’d have to stop Synapse and start the other web server every time we renewed our certificate, and that means downtime. Instead, let’s put our Synapse behind a proper web server and let that serve the files. This has added advantages, including that we can host our Matrix home server on the standard port 443 without having to run Synapse as root. For this example, we’re going to use NGINX, so start by installing NGINX in whatever way your Linux distribution of choice recommends. Now, you should have a webroot for your new web server somewhere. Hopefully your helpful Linux distribution has started you off with a config file – let’s see: <pre># nano /etc/nginx/nginx.conf
 </pre> We’re looking for the ‘server’ section of that file. We need to make it look something like this: <pre> server {

```
      # Make sure this is 0.0.0.0: no use listening on 127.0.0.1 or we'll
only be&#13;
      # serving to ourselves! There's no port here, which means we'll listen
on&#13;
      # port 80&#13;
      listen 0.0.0.0;&#13;
```



```
      server_name example.com www.example.com;&#13;
```



```
      access_log /var/log/nginx/example.com.access_log main;&#13;
      error_log /var/log/nginx/example.com info;&#13;
```



```
      # This is where we put the files we want on our site&#13;
      root /var/www/examplecom/htdocs;&#13;
```



```
      # Here's where it gets interesting: This will send any path that
starts&#13;
      # with /_matrix to our Synapse!&#13;
```



```
      location /_matrix {&#13;
          proxy_pass http://localhost:8008;&#13;
      }&#13;
  }&#13;
```

</pre> When you’re happy with the look of that file, let’s restart the server: <pre># nginx -s reload
 </pre> Before we go any further, let’s test our new configuration: <pre>$ curl http://example.com/_matrix/key/v2/server/auto
 {„old_verify_keys“:{},„server_name“:„example.com“,„signatures“:{„example.com“:{„ed25519:auto“ :„RWb+w6vHUUokoDgElwG6Cg50ezZvBrzXtJmJIH8jEwI5x0JQ7prn3FwjhbgKTH5jE7J8lly3HEc4COn4JCC vCA“}},„tls_fingerprints“:[{„sha256“:„DMbzSZ5Uj7/6p/RT/UtQYJLHm5o0TwBSVYXsqpDdVDs“}],„valid _until_ts“:1455203001035,„verify_keys“:{„ed25519:auto“:{„key“:„1YiTDjmE86AlmrbIYE2lyqauV9wPo 8jw2kxZAZFfl/Q“}}}
 </pre> Those are your server’s public keys! Now we have a web server running, we can get our SSL certificate. Let’s Encrypt have their own client which will automate everything including rewriting your NGINX config file, however that means it has a large number of dependencies and needs to be run as root. For this example, we’re going to use the much simpler acme_tiny.py. I’m going to assume you have a user called, ‘letsencrypt’, so, as root, let’s set up the place for it to write its challenge files: <pre># mkdir /var/www/examplecom/htdocs/.well-known/acme-challenge
 # chown letsencrypt:users /var/www/examplecom/htdocs/.well-known/acme-challenge
 </pre> Now let’s switch to our letsencrypt user: <pre>$ ssh letsencrypt@example.com
 </pre> We’ll start by getting ourselves a copy of acme_tiny.py: <pre>$ git clone https://github.com/diafygi/acme-tiny.git
 </pre> Now let’s set up a directory structure (let’s say we might want to manage more than one domain someday): <pre>$ mkdir examplecom
 $ cd examplecom
 $ ln -s /var/www/examplecom/htdocs/.well-known/acme-challenge challenges
 </pre> Now, we’ll need to generate two keys for Let’s Encrypt, and account key and a domain key. The former is what we use to identify ourselves to Let’s Encrypt and the latter is the key we use to do the actual SSL. <pre>$

openssl genrsa 4096 > letsencrypt_examplecom_account.key
 Generating RSA private key, 4096 bit long modulus
 ..++
.....
.....
.....++
 e is 65537 (0x10001)
 $ chmod 600 letsencrypt_examplecom_account.key
 $ openssl genrsa 4096 > letsencrypt_examplecom_domain.key
 Generating RSA private key, 4096 bit long modulus
 .....++
.....
.....++
 e is 65537 (0x10001)
 $ chmod 600 letsencrypt_examplecom_domain.key
 </pre> Now, store those keys somewhere safe! After you’ve done that, let’s generate a certificate request for our domain. Note that we’re requesting one for both example.com and www.example.com: this isn’t strictly necessary for Matrix but could be useful if we want to host a website too. <pre>$ openssl req -new -sha256 -key letsencrypt_examplecom_domain.key -subj „/“ -reqexts SAN -config <(cat /etc/ssl/openssl.cf <(printf „[SAN]\nsubjectAltName=DNS:example.com,DNS:www.example.com“)) > examplecom.csr
 </pre> Okay, we have our keys, our certificate request, and somewhere to host our challenge files, so we’re ready to request a certificate! Be careful about this part and make sure you’ve got everything right, because Let’s Encrypt enforce strict rate limits on the number of certificates you can request for one domain. Here we go: <pre>$ python ~/acme-tiny/acme_tiny.py –account letsencrypt_examplecom_account.key –csr examplecom.csr –acme-dir challenges/ > examplecom.crt
 Parsing account key…
 Parsing CSR…
 Registering account…
 Registered!
 Verifying example.com…
 example.com verified!
 Verifying www.example.com…
 www.example.com verified!
 Signing certificate…
 Certificate signed!
 </pre> Is that it, did it work? Well, let’s see: <pre>$ openssl x509 -in examplecom.crt -noout -text
 Certificate:

```
  Data:&amp;#13;
      Version: 3 (0x2)&amp;#13;
      Serial Number:&amp;#13;
          01:02:22:77:02:1b:eb:d5:3d:c3:14:6d:87:43:22:3d:fc:0f&amp;#13;
  Signature Algorithm: sha256WithRSAEncryption&amp;#13;
      Issuer: C=US, O=Let's Encrypt, CN=Let's Encrypt Authority X3&amp;#13;
      Validity&amp;#13;
          Not Before: Feb  6 21:37:00 2016 GMT&amp;#13;
          Not After : May  6 21:37:00 2016 GMT&amp;#13;
      Subject: CN=example.com&amp;#13;
      Subject Public Key Info:&amp;#13;
```

[etc]
 </pre> Congratulations, you have an official, signed certificate for your domain! Now, before we can use it, we need to add the Let’s Encrypt certificate to it, because our web server needs to send both: <pre>$ wget https://letsencrypt.org/certs/lets-encrypt-x3-cross-signed.pem
 –2016-02-06 23:38:55– https://letsencrypt.org/certs/lets-encrypt-x3-cross-signed.pem
 Resolving letsencrypt.org… 23.66.17.98, 2a02:26f0:60:489::2a1f, 2a02:26f0:60:481::2a1f
 Connecting to letsencrypt.org|23.66.17.98|:443… connected.
 HTTP request sent, awaiting response… 200 OK
 Length: 1675 (1.6K) [application/x-x509-ca-cert]
 Saving to: ‘lets-encrypt-x3-cross-signed.pem’
 lets-encrypt-x3-cross-signed.pe 100%[==>] 1.64K –.-KB/s in 0s
 2016-02-06 23:38:55 (61.5 MB/s) - ‘lets-encrypt-x3-cross-

signed.pem’ saved [1675/1675]
 $ cat examplecom/examplecom.crt lets-encrypt-x3-cross-signed.pem >examplecom/examplecom_cert_chain.crt
 </pre> Now’s let’s symlink it in place, along with the domain key, so we can renew it easily later. We’ll need to be root again for this: <pre>$ ssh root@example.com
 # ln -s /home/letsencrypt/examplecom/examplecom_cert_chain.crt /etc/ssl/nginx/examplecom_cert.pem
 # ln -s /home/letsencrypt/examplecom/letsencrypt_examplecom_domain.key /etc/ssl/nginx/examplecom_key.pem
 </pre> Now, one more crucial thing we have to do before using our SSL is to give NGINX some Diffie Hellman parameters. This is a good thing to do for any SSL configuration (it will increase your score on SSL Labs) but it’s absolutely crucial for us because Synapse will only negotiate forward secret connections, so otherwise other Matrix home servers will refuse to talk to us! (Technically, Synapse also support elliptic curve Diffie Hellman, which doesn’t need DH parameters, but not all Synapses will support this.) You’ll already have some Diffie Hellman parameters from you existing Synapse, so you could use them: <pre># cp /home/synapse/synapse/matrix.example.com.tls.dh /etc/ssl/nginx/examplecom_dhparams.pem
 </pre> …or you can generate your own. You’ll probably want to do this on your desktop or laptop if you have OpenSSL installed, it will be much faster: <pre>$ openssl dhparam -out examplecom_dhparams.pem 2048
 Generating DH parameters, 2048 bit long safe prime, generator 2
 This is going to take a long time
 .....+.....[etc, etc]
 $ scp examplecom_dhparams.pem root@example.com:/etc/ssl/nginx/examplecom_dhparams.pem
 </pre> Now, let’s get our new certificate in action! Open up your NGINX config file again, and add another server block that look like this: <pre> server {

```
      listen 0.0.0.0:443;&amp;#13;
      server_name example.com www.example.com;&amp;#13;
      ssl on;&amp;#13;
      ssl_certificate /etc/ssl/nginx/examplecom_crt.pem;&amp;#13;
      ssl_certificate_key /etc/ssl/nginx/examplecom_key.pem;&amp;#13;
      ssl_dhparam /etc/ssl/nginx/examplecom_dhparams.pem;&amp;#13;
      ssl_protocols TLSv1 TLSv1.1 TLSv1.2;&amp;#13;
      ssl_prefer_server_ciphers on;&amp;#13;
      # mozilla intermediate list, jan 2016&amp;#13;
      ssl_ciphers "ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-
SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-
AES128-GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-
SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-
SHA:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-
SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-
AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-DSS-AES256-SHA:DHE-RSA-AES256-
SHA:ECDHE-RSA-DES-CBC3-SHA:ECDHE-ECDSA-DES-CBC3-SHA:EDH-RSA-DES-CBC3-
SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:AES128-
SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-
SHA:!aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-
SHA:!KRB5-DES-CBC3-SHA";&amp;#13;
      ssl_session_cache shared:SSL:50m;&amp;#13;
      access_log /var/log/nginx/examplecom.ssl_access_log main;&amp;#13;
```

```
      error_log /var/log/nginx/examplecom.ssl_error_log info;&#13;
      root /var/www/examplecom/htdocs;&#13;
      location /_matrix {&#13;
          proxy_pass http://localhost:8008;&#13;
      }&#13;
  }&#13;
```

</pre> It looks pretty similar to our previous server block, except for all that stuff about SSL in the middle. We’re pointing NGINX at our certificate, key and Diffie Hellman parameter files and specifying what protocols and ciphers we want our server to talk. The long list here is taken from Mozilla’s Server Side TLS guidelines and is their ‘Intermediate’ list. See that page for more information on what that means, and choose a different list of ciphers if you prefer: just remember we must support at least the ephemeral Diffie Hellman ciphers, or other home servers won’t talk to us! Now let’s restart our NGINX and see if it works: <pre># nginx -s reload
 </pre> …and that command again, this time with https: <pre>$ curl https://example.com/_matrix/key/v2/server/auto
 {„old_verify_keys“:{},„server_name“:„example.com“,„signatures“:{„example.com“:{„ed25519:auto“ :„RWb+w6vHUUokoDgElwG6Cg50ezZvBrzXtJmJIH8jEwI5x0JQ7prn3FwjhbgKTH5jE7J8lly3HEc4COn4JCC vCA“}},„tls_fingerprints“:[{„sha256“:„DMbzSZ5Uj7/6p/RT/UtQYJLHm5o0TwBSVYXsqpDdVDs“}],„valid _until_ts“:1455203001035,„verify_keys“:{„ed25519:auto“:{„key“:„1YiTDjmE86AlmrbIYE2lyqauV9wPo 8jw2kxZAZFfl/Q“}}}
 </pre> Hooray! You should now be able to open a browser to https://example.com/matrix/ and log in securely over SSL! <h2 class=„p1“>Renewing Your Certificate</h2> Now, there’s one important step left, and that’s to set up renewal for the certificate, otherwise we’ll find our shiny new SSL will stop working in three months time. We can use the same acme_tiny command to do this: <pre>$ python ~/acme-tiny/acme_tiny.py –account letsencrypt_examplecom_account.key –csr examplecom.csr –acme-dir challenges/ > examplecom.crt
 Parsing account key...
 Parsing CSR...
 Registering account...
 Already registered!
 Verifying example.com...
 example.com verified!
 Verifying www.example.com...
 www.example.com verified!
 Signing certificate...
 Certificate signed!
 $ wget https://letsencrypt.org/certs/lets-encrypt-x3-cross-signed.pem
 –2016-02-06 23:38:55– https://letsencrypt.org/certs/lets-encrypt-x3-cross-signed.pem
 Resolving letsencrypt.org... 23.66.17.98, 2a02:26f0:60:489::2a1f, 2a02:26f0:60:481::2a1f
 Connecting to letsencrypt.org|23.66.17.98|:443... connected.
 HTTP request sent, awaiting response... 200 OK
 Length: 1675 (1.6K) [application/x-x509-ca-cert]
 Saving to: ‘lets-encrypt-x3-cross-signed.pem’
 lets-encrypt-x3-cross-signed.pe 100%[==========>] 1.64K –.-KB/s in 0s
 2016-02-06 23:38:55 (61.5 MB/s) - ‘lets-encrypt-x3-cross-signed.pem’ saved [1675/1675]
 $ cat examplecom/examplecom.crt lets-encrypt-x3-cross-signed.pem >examplecom/examplecom_cert_chain.crt
 </pre> Synapse will automatically pick up the new certificate, but we’ll need to tell NGINX to reload: <pre># nginx -s reload
 </pre> Setting up a cronjob to automate this is left as an exercise to the reader! <h2 class=„p1“>Federation behind the HTTP Proxy</h2> If you like, you can stop reading now: our clients can access our home server securely but other home server are still talking to our Synapse directly on port 8448. This is fine, and if you’re happy with this, you can stop reading now. But remember how we made sure other Synapses could talk to our NGINX? Well, why not put federation behind our new web server too? Now, we need to do a couple of things to make this work: were you looking carefully at the JSON those curl commands returned? If you were, you might have noticed a key

called, ‘tls_fingerprints’. Our home server serves up a fingerprint of the TLS certificate its using from this API, and we’ve just given our web server a different certificate, so we need to give Synapse our new certificate. How are we going to tell other home servers to talk to our NGINX instead? Well, ultimately we’re going to change our DNS SRV record to point at port 443 instead of port 8448, but that change could take a while to propagate through caches, so let’s test it by having our NGINX listen on port 8448 temporarily. We can do this by copying that same block from above, but with a different port: <pre> server {

```
      listen 0.0.0.0:8448;&amp;#13;
      server_name example.com www.example.com;&amp;#13;
  [etc]&amp;#13;
```

</pre> Don’t restart NGINX just yet: we need to tell our Synapse to stop listening on that port first, so lets do that and give it our new certificate: <pre>$ nano /home/synapse/synapse/homeserver.yaml
 </pre> Now we’ll want to find and edit the following lines: <pre>tls_certificate_path: „/etc/ssl/nginx/examplecom_crt.pem“
 # We can comment this out, as long as we set no_tls to true below
 # tls_private_key_path: „/whatever/path/synapse/generated“
 # PEM dh parameters for ephemeral keys
 tls_dh_params_path: „/whatever/path/synapse/generated“
 # Turn off TLS everywhere (this overrides the listeners section below)
 no_tls: True

1. port: 8008

tls: false

```
  # We can bind to only localhost since only the local nginx needs to hit
this&amp;#13;
  bind_address: '127.0.0.1'&amp;#13;
  type: http&amp;#13;
  # Set this so that Synapse obeys nginx's X-Forwarded-For headers, then IP
addresess will be correct in Synapse's logs&amp;#13;
  x_forwarded: true&amp;#13;
  resources:&amp;#13;
    - names: [client, webclient]&amp;#13;
      compress: true&amp;#13;
    - names: [federation]&amp;#13;
      compress: false&amp;#13;
```

</pre> Note: if you have an old enough config file that you have ‘bind_host’ and ‘bind_port’ directives, now is the time to remove them. Now let’s restart Synapse and our web server to swap over what’s listening on our port 8448: <pre>$ synctl restart
 # nginx -s reload
 </pre> Now let’s try that test again on port 8448: <pre>$ curl https://example.com:8448/_matrix/key/v2/server/auto
 {„old_verify_keys“:{},„server_name“:„example.com“,„signatures“:{„example.com“:{„ed25519:auto“ :„bdca31805e4209f6ff4d644251a29d0cb1dc828a4d6131c57cf8305288f337c0“}},„tls_fingerprints“:[{ „sha256“:„1d9ec66599e229654a79f28e26675fdeb585027553af6d581926e821a6b6527c“}],„valid_u ntil_ts“:1455203001035,„verify_keys“:{„ed25519:auto“:{„key“:„1YiTDjmE86AlmrbIYE2lyqauV9wPo8j

w2kxZAZFfl/Q"}}}
 </pre> Notice anything different? The tls_fingerprints part has changed because we now have a different certificate. The signatures/example.com/ed25519:auto value has changed too: that’s because that part is a signature of the rest of JSON object, so changing the tls_fingerprints has caused this to change too. And that’s it! If you’re happy everything is working, you can then change your DNS SRV record to point at port 443 instead of 8448, then after leaving a few days for the change to propagate through caches, remove the extra server block from your nginx.conf and restart to stop your nginx listening on port 8448.

```
                </div><div alt="article">
                <p>Historically we&amp;#8217;ve had two projects for iOS:</p>
```

MatrixSDK: a low level library to interact with a Matrix homeserver Console: an example Matrix client based on MatrixSDK The primary intention of Console was to demonstrate how to use MatrixSDK to write a Matrix client app.
However, this split isn’t helpful for developers who want higher level modules that provides UIViewControllers ready to use in an existing app, with no need to manage low level communications with the Matrix homeserver. It is where the MatrixKit project started. MatrixKit sits between MatrixSDK and your existing iOS app. It provides customisable UIViewControllers a developer can integrate in their app. If you want to add to your app a screen to chat in a room, you just need to use the MXKRoomViewController. We made MatrixKit so that the components it provides are easy to integrate but also easy to customise. We do not have yet full samples of customisation as we’ve been focused on the library core, but here are a few examples:
    You probably recognise the theme of the first one, as it’s what we use in the Console app today.
The second one is the iOS7-style look and feel from JSQMessagesViewController. With few lines of code we connected it to MatrixKit data models. Yes, data models provided by MatrixKit are reusable too. MatrixKit is also highly extensible. If you want to create new table cells to render messages, new views, new view controllers, etc, you will find a place to hook them into the MatrixKit code. MatrixKit repository is here: https://github.com/matrix-org/matrix-ios-kit and it is available via CocoaPods (the MatrixKit pod). In parallel of MatrixKit, we did some spring-cleaning – the official Matrix.org iOS offerings are now split into three github repos. One for each deliverable: https://github.com/matrix-org/matrix-ios-

sdk https://github.com/matrix-org/matrix-ios-kit https://github.com/matrix-org/matrix-ios-console Other releases: Today, we released MatrixSDK 0.4.0 (changes). Update your pods :) Console 0.4.0 (changes) is in the Apple submission process. This will be the first version of the app using MatrixKit. Aesthetically, there is no change since the previous version. The app is more stable due to all the data abstractions and management improvements provided by MatrixKit. If you’re an iOS developer, please have a go with MatrixKit and let us know on #ios:matrix.org how you get on!

```
                </div><div alt="article">
                <p>Hi all,</p>
```

What with the chaos of Mobile World Congress last week we seem to have been hoarding releases – so here’s what’s been happening! Synapse 0.8.0 was released this afternoon. This is a major performance/stability release, with lots of good stuff going on – especially adding more customisable push notification support APIs for iOS/Android; support for registering accounts from mobile devices; extensions to the new Application Service API and lots of federation improvements and bug fixes. Please upgrade at your earliest convenience. Meanwhile, we quietly released the Matrix Console Android example app to the Google Play last week, alongside v0.2.2 of the Android SDK – release notes below. There’ll be a new version of the Android Console app out tomorrow, but mentioning here for completeness and to share the release notes. Also, the iOS SDK is now on v0.3.1, with lots of performance and usability improvements. Finally, we have a whole new official Matrix client SDK for JavaScript, all packaged up ready for use by Node developers and JS purists alike as an NPM with minimal dependencies. Meanwhile, the matrix-angular-sdk has been switched to use matrix-js-sdk from now on. You can use the plain JS API with a

```
npm install matrix-js-sdk
```

and then: <pre>
 var sdk = require(„matrix-js-sdk“);
 var client = sdk.createClient(„https://matrix.org“);
 client.publicRooms(function(err, data) {

```
  console.log("Public Rooms: %s", JSON.stringify(data));&amp;#13;
```

});

 </pre> Meanwhile, release notes for all & sundry lie below. <pre>

 Changes in synapse v0.8.0 (2015-03-06)

==========

 General:

 * Add support for registration fallback. This is a page hosted on the server

```
which allows a user to register for an account, regardless of what
client&#13;
they are using (e.g. mobile devices).&#13;
```


 * Added new default push rules and made them configurable by clients:

- Suppress all notice messages.
- Notify when invited to a new room.
- Notify for messages that don't match any rule.
- Notify on incoming call.

 Federation:

 * Added per host server side rate-limiting of incoming federation requests.
 * Added a ``/get_missing_events/`` API to federation to reduce number of

```
``/events/`` requests.&#13;
```


 Configuration:

 * Added configuration option to disable registration:

```
``disable_registration``.&#13;
```

* Added configuration option to change soft limit of number of open file

```
descriptors: ``soft_file_limit``.&#13;
```

* Make ``tls_private_key_path`` optional when running with ``no_tls``.

 Application services:

 * Application services can now poll on the CS API ``/events`` for their events,

```
by providing their application service ``access_token``.&#13;
```

* Added exclusive namespace support to application services API.

 </pre> <pre>
 Changes in Matrix Android SDK in 0.2.2 (2015-02-27)
 ==========

 —-
 SDK
 —-

```
&#13;
```

———–
 Matrix Console
 ———–
 Improvements:
 * Exif management : the uploaded image is rotated according to the exif metadata

```
 (if the device has enough free memory).&#13;
```

* Add a piechart while downloading an image
 * Add JSON representation of a message (tap on its row, “Message details”
 * The public rooms list is now sorted according to the number of members.

 Features:
 * Add configuration and skeleton classes for receiving GCM messages
 * Add REST client for pushers API with add method for HTTP pushers.
 * Add the account creation.

 Bug fixes:
 * Reset the image

thumbnail when a row is reused.
 * SYAND-30 Notification should be away when entering a room.
 * Some images thumbnails were downloaded several times.
 * Restore the foreground service
 * The medias cache was not cleared after logging out.
 * The client crashed when joining #anime:matrix.org.
 * SYAND-29 Messages in delivery status are not seen
 * Some user display names were their matrix IDs.
 * The room name/ topic were invalid when inviting to a room.

 </pre> <pre>
 Changes in Matrix iOS SDK in 0.3.1 (2015-03-03)
 ==========

 —–
 SDK
 —–
 Improvements:
 * Improved push notifications documentation.
 * MXSession: Slightly randomise reconnection times by up to 3s to prevent all

```
 Matrix clients from retrying requests to the homeserver at the same
time.&amp;#13;
```

* Improved logs

 Bug fixes:
 * SYIOS-90 - iOS can receive & display messages multiple times when on bad connections

 ——————–
 Matrix Console
 ——————–
 Improvements:
 * Fixed warnings with 64bits builds.
 * Room history: Improve scrolling handling when keyboard appears.
 * Contacts: Prompt user when local contacts tab is selected if constact sync is disabled.

 Bug fixes:
 * Fix crash when switching rooms while the event stream is resuming.
 * SYIOS-69 - On Screen Keyboard can end up hiding the most recent messages in a room.
 * SYIOS-98 - Crash when attempting to attach image on iPad

 Changes in Matrix iOS SDK in 0.3.0 (2015-02-23)
 ==========

 —–
 SDK
 —–
 Breaks:
 * [MXSession initWithMatrixRestClient: andStore:] and the onStoreDataReady argument in

```
 [MXSession start:] has been removed. The SDK client can now use the
asynchronous&amp;#13;
 [MXSession setStore:] method to define a store and getting notified when
the SDK can&amp;#13;
 read cached data from it. (SYIOS-62)&amp;#13;
```

* MXStore implementations must now implement [MXStore openWithCredentials].
 * All MXRestClient methods now return MXHTTPOperation objects.

 Improvements:
 * Created the MXSession.notificationCenter component: it indicates when an event must be

```
 notified to the user according to user's push rules settings.&amp;#13;
```

* MXFileStore: Improved loading performance by 8x.
 * Added an option (MXSession.loadPresenceBeforeCompletingSessionStart) to refresh presence

```
 data in background when starting a session.&amp;#13;
```

* Created MXLogger to redirect NSLog to file and to log crashes or uncaught exception.
 * MXRestClient: Added [MXRestClient registerFallback].
 * Logs: Make all NSLog calls follows the same format.

 Features:
 * SYIOS-40 - Any HTTP request can fail due to rate-limiting on the server, and need to be retried.
 * SYIOS-81 - Ability to send messages in the background.

 Bug fixes:
 * SYIOS-67 - We should synthesise identicons for users with no avatar.
 * MXSession: Fixed crash when closing the MXSession before the end of initial

Sync.
 </pre>

```
                </div><div alt="article">
                <p><strong>This post has now been edited into a guide
&amp;#8211; you can find <a
href="https://github.com/matrix-org/matrix-doc/blob/master/supporting-docs/g
uides/2015-08-21-application_services.md">the source in github</a>, and the
<a href="http://matrix.org/docs/guides/application_services.html">formatted
guide on the matrix.org website</a>!</strong></p>
```

<hr/> Hi everyone. I’m Kegan, one of the core developers on Matrix. I’d like to explain a bit more about one of the upcoming features in Matrix: Application Services. This is an entirely new component in the Matrix architecture which gives great power (along with great responsibility!) to the wielder. Application services are distinct modules which which sit alongside a home server providing arbitrary extensible functionality decoupled from the home server 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. One of the main use cases for application services is for protocol bridges. As you may know, we have an IRC bridge bot on matrix.org which resides as a user on #matrix, #matrix-dev, #openwebrtc and #vuc which bridges into freenode. There is nothing special about this bot; it is just treated as a client. However, this limits the things the bot can do. For example, the bot cannot reserve the virtual user IDs it creates, and cannot lazily bridge arbitrary IRC rooms on-the-fly. By using application services, the IRC bot can do both of these things. This would allow any Matrix user to join a room alias which doesn’t exist: say

```
#irc.freenode.python:matrix.org
```

, which would then tell the application service to create a new Matrix room, make the alias for it, join #python on freenode and bridge into it. Any IRC user on #python would then be provisioned as a virtual user e.g.

```
@irc.freenode.alice:matrix.org
```

. This also allows PMs to be sent directly to

```
@irc.freenode.alice:matrix.org
```

, no matter what channel Alice is on. 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 home servers. Only home server admins can allow an application service to link up with their home server, and the application service is in no way federated to other home servers. You can think of application services as additional logic on the home server itself, without messing around with the book-keeping that home servers have to do. This makes adding useful functionality very easy. Example The application service (AS) API itself uses webhooks to communicate from the home server to

the AS: – Room Alias Query API : The home server hits a URL on your application server to see if a room alias exists.
– User Query API : The home server hits a URL on your application server to see if a user ID exists.
– Push API : The home server hits a URL on your application server to notify you of new events for your users and rooms. A very basic application service may want to log all messages in rooms which have an alias starting with “#logged_” (side note: logging won’t work if these rooms are using end-to-end encryption). Here’s an example of a very basic application service using Python (with Flask and Requests) which logs room activity: <pre># app_service.py:

 import json, requests # we will use this later
 from flask import Flask, jsonify, request
 app = Flask(name)

 @app.route(„/transactions/<transaction>“, methods=[„PUT“])
 def on_receive_events(transaction):

```
  events = request.get_json()["events"]&#13;
  for event in events:&#13;
      print "User: %s Room: %s" % (event["user_id"], event["room_id"])&#13;
      print "Event Type: %s" % event["type"]&#13;
      print "Content: %s" % event["content"]&#13;
  return jsonify({})&#13;
```


 if name == „main“:

```
  app.run()&#13;
```

</pre> Set your new application service running on port 5000 with: <pre>python app_service.py
 </pre> The home server 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’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. <pre># homeserver.yaml
 app_service_config_files:

1. „/path/to/appservice/registration.yaml“

</pre> NB: Note the “-” at the start; this indicates a list element. The registration file

```
registration.yaml
```

should look like: <pre># registration.yaml

 # 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: []&#13;
```

```
rooms: []&#13;
aliases:&#13;
  - exclusive: false&#13;
    regex: "#logged_.*"&#13;
```

</pre> You will need to restart the home server after editing the config file before it will take effect. To test everything is working correctly, go ahead and explicitly create a room with the alias “#logged_test:localhost” and send a message into the room: the HS will relay the message to the AS by PUTing to /transactions/<tid> and you should see your AS print the event on the terminal. This will monitor any room which has an alias prefix of “#logged_”, but it won’t lazily create room aliases if they don’t already exist. This means it will only log messages in the room you created before: #logged_test:localhost. Try joining the room “#logged_test2:localhost” without creating it, and it will fail. Let’s fix that and add in lazy room creation: <pre>@app.route(„/rooms/<alias>“)
 def query_alias(alias):

```
  alias_localpart = alias.split(":")[0][1:]&#13;
  requests.post(&#13;
      # NB: "TOKEN" is the as_token referred to in registration.yaml&#13;
      "<a
href="http://localhost:8008/_matrix/client/api/v1/createRoom?access_token=TO
KEN"
target="_blank">http://localhost:8008/_matrix/client/api/v1/createRoom?acces
s_token=TOKEN</a>",&#13;
      json.dumps({&#13;
          "room_alias_name": alias_localpart&#13;
      }),&#13;
      headers={"Content-Type":"application/json"}&#13;
  )&#13;
  return jsonify({})&#13;
```

</pre> 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 “#logged_test2:localhost” and it will work as you’d expect. 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. Application services are powerful components which extend the functionality of home servers, but they are limited. They can only ever function in a “passive” way. 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 fact that censoring will not work when using end-to-end encryption, all federated home servers would also need to reject the event in order to stop developing an inconsistent event graph. To “actively” monitor events, another component called a “Policy Server” is required, which is beyond the scope of this post. 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. If you are bridging huge amounts of traffic, you may be better off having your bridge directly talk the Server-Server federation API rather than the simpler Application Service API. I hope this post 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 section in the spec, or the raw drafts and spec in https://github.com/matrix-org/matrix-doc/. The AS API is not yet frozen, so feedback is very welcome!

```
                </div><div alt="article">
                <p>One of the final remaining missing bits of Matrix today
is specifying and implementing the Application Service (AS) APIs which allow
you to easily extend Matrix with custom server-side functionality. The AS
APIs should let you perform any arbitrary manipulation on chatroom contents,
modulo end-to-end encryption constraints &amp;#8211; e.g. machine translation;
archiving/searching contents; interactive automated services; conferencing;
firing push notifications and other hooks; etc. If you really want to look
behind the curtain, the bug tracking the development (somewhat out-of-date)
is at <a href="https://matrix.org/jira/browse/SPEC-34">SPEC-34</a>.</p>
```

However, the most obvious use case for this is gatewaying Matrix rooms through to existing communication platforms (XMPP, SIP, non-standardised systems) – which is obviously key to Matrix’s overall goal of defragmenting communication. And the good news is that even though the AS APIs don’t yet exist, we can still make good progress here through the existing client-server API. Anyone who’s hung around chat systems like IRC should be familiar with the idea of bots – non-human users who participate in chatrooms and but perform arbitrary automated functionality – and you can go quite a long way in using the ‘bot’ idiom to add automatic functionality to Matrix. [In fact the first AS API we’ll be adding will probably be simply extending the client-server API with some additional privileges to allow homeserver admins to hook up services to their server which are essentially privileged bots (e.g. ability to autojoin rooms created on that server with high power-level; ability to flag themselves as an invisible ‘service bot’; ability to monitor events from rooms without joining them: useful for read-only services such as sending push notifications, adding search/archive functionality; etc). This should also be familiar to IRC users, as it’s similar to the model that IRC Services uses.] So, we already have a few bots hanging around prototyping out bridging to other systems, which hopefully should evolve into full Application Services (where it makes sense; sometimes a bot is good enough). For instance, we have the Matrix/IRC Bridge thanks to tm604 and LeoNerd. The way this works is simply a bot which joins IRC channels and their Matrix room equivalents; watching the messages on both sides of the bridge and relaying them to the other side, creating virtual users as required. In future we can be smarter about having the bridge talk on behalf of actual users, or letting actual users control their virtual users, but it’s good enough as a first cut. So for Friday’s VUC 517, we decided at the last minute (on Tuesday) to make as much of VUC accessible via Matrix as possible. One side of this was hooking up the Jitsi Video Bridge to be accessible by talking to the underlying XMPP server – the other side was hooking up via SIP to the ZipDX audio bridge that is used for audio-only participants in the conference. Both of these would be done as Matrix bots – a virtual user that you could voice/video call 1:1 over Matrix which would then route your call into VUC appropriately. By Thursday, the Jitsi bot got to the point of being able to place calls and see a single video stream (picked at random), but the video uplink wasn’t getting through and actually selecting the right stream to watch (or viewing multiple

streams) wasn’t in place either. I’m sure there’ll be a similar blogpost once it’s finished, so I’m not going to talk about it further here. Meanwhile, on Thursday night we hacked together a separate bot to jack into the ZipDX bridge via SIP. Tim Panton’s suggestion was to use FreeSWITCH‘s mod_verto for this, and after Anthony Minesalle and Mike Jerris from FreeSWITCH had popped up on #matrix:matrix.org on Tuesday to find out what we’re up to, this seemed like serendipity. We hadn’t played with mod_verto before, although had been pointed at it by someone on IRC shortly after we released Matrix – it’s a cool project from the FreeSWITCH dev team that exposes a simple JSON-RPC interface for call signalling over websockets, providing a much more suitable way for WebRTC developers to get calls in and out of FreeSWITCH than shoehorning a SIP stack into their browser. In this respect it’s quite similar to Matrix, but there are some huge differences: Verto doesn’t (yet) do federation – either for message-passing (like XMPP) or history-replication (like Matrix or XMPP FMUCs). In fact, Matrix fundamentally competes more with JSON-RPC at OSI layer 5 by defining a standardised RESTful API for federated state synchronisation – which so happens to define some datatypes for VoIP signalling; whereas Verto currently seems to be focused solely on the application-layer problem of VoIP signalling. Verto is a generic RPC API with method names like verto.invite, verto.answer, verto.media, verto.bye etc. for handling call signalling. It’s obviously readily extensible to any other functionality expressed as an RPC invocation. The Matrix client-server API however passes around event objects within the context of a room – it’s message passing rather than RPC. Matrix’s VoIP events support trickle-ICE; announcing ICE candidates from WebRTC as and when they become available. This good is for speedy call-setup (you don’t have to wait for all ICE to complete before setting up the call) and to support call continuity when roaming between different IP networks (in theory). However, Verto currently requires ICE candidates to be presented monolithically – it hasn’t made sense to implement trickle-ICE as FreeSWITCH’s core engine doesn’t support it yet. Verto looks to be wired to speak JSON-RPC over Websockets, whereas Matrix deliberately uses plain old HTTP as its baseline for maximum simplicity and compatibility in PUTting and GETting JSON events around Verto could be an interoperable standard but the API hasn’t been documented yet (unless I’ve totally missed it) – to build the bot we looked at the websockets tab in Chrome’s network inspector and drew some inferences from the JSON seen in a call placed using the FreeSWITCH Verto demo site, which was very intuitive and almost self-documenting. Meanwhile, the (minimal) doc for Matrix’s events is up at http://matrix.org/docs/spec/#voice-over-ip. Verto has a huge advantage however, in that FreeSWITCH has a mod_verto today, and doesn’t have a mod_matrix – so one can use mod_verto right now as an easy way to get VoIP calls in and out of FreeSWITCH from the web and bridge them to SIP! So, when writing a Matrix<->SIP bridging bot for VUC, Verto turned out to be a really nice way to quickly get something up and running. The end result is at https://github.com/matrix-org/synapse/blob/develop/contrib/vertobot/bot.pl – a (precisely!) 300 line perl script built on LeoNerd’s Net-Async-Matrix and Net::Async::Protocol::WebSock

et which logs into Matrix and routes any 1:1 audio calls it receives through to the defined mod_verto service. Currently it gleefully hardcodes in the destination extension it calls and the caller-ID, but this could trivially be extended. It also chokes on SSL WebSockets, so your mod_verto needs to be listening unencrypted on port 8081. The task of mapping between Matrix m.call.* VoIP events and Verto verto.* RPC methods was almost entirely trivial (although I hasten to add that Matrix’s and Verto’s were developed completely independently – it’s just that there are only so many ways to express VoIP signalling in JSON!) Matrix’s m.call.invite is equivalent to the combination of verto.invite + verto.media (but may lack ICE candidates) Matrix’s m.call.candidates has no direct equivalent, so has to be coalesced and merged into verto.media Matrix’s m.call.answer is equivalent to verto.answer (but may lack ICE candidates) Matrix’s m.room.displayname is equivalent to verto.display (assuming I understand verto.display) Matrix’s m.call.hangup is equivalent to verto.bye …and these are the only verto RPCs we mapped. For the demo itself, we obviously needed a FreeSWITCH with mod_verto all up and running to hook into the ZipDX bridge: our friends at Truphone were good enough to provide one at zero notice (Thanks James, Andy, Giacomo!), and we were up and running. Unfortunately we did hit some problems: Net::Async::Matrix has a few quirks which LeoNerd is working out currently; the bot doesn’t coalesce the trickle-ICE properly currently causing a race-condition that means ICE setup may fail; Matthew’s use of IO::Async was a bit buggy; and moreover we didn’t have time to implement DTMF… which was a bit of a shame as you can only unmute yourself on the ZipDX bridge via DTMF *5! But in general, the mini-hackathon was a success and it was great fun to be able to listen into VUC via the bridge and demonstrate the first ever Matrix<->SIP call! The bot ran as @vucbot:matrix.org, although is turned off now as there’s no VUC to listen to, and the FreeSWITCH & bot were only deployed temporarily. Once the kinks mentioned above are sorted out we’ll hopefully set it running again permanently! And hopefully this little bot is an exciting precursor to more robust Matrix bridges and application services in the months to come… If you’re crazy enough to want to try to run the bot yourself, then it should actually be quite simple to get up and running:

```
 # grab synapse if you don't have it already<br/>git clone
https://github.com/matrix-org/synapse.git synapse-develop<br/>cd synapse-
develop
```

 # you'll need the develop branch, as we haven't released a build with vertobot in it yet
git checkout develop
cd contrib/vertobot # you'll need cpanm to install the perl dependencies
cpan -i App::cpanminus
cpanm –installdeps . # manually install a develop version of Net::Async::Matrix as cpanm can't figure it out, seemingly
cpanm –force PEVANS/Net-Async-Matrix-0.11_002
(you may need to also replace the 'croak' for the „Already have a room with ID“ error with 'warn' in Net::Async::Matrix if the bot crashes with this error) # create a username account for your bot on a Matrix homeserver somewhere at this point # set up a config file
cp config.yaml mybot.yaml
edit mybot.yaml to taste - at the least you must specify the login & password & homeserver for your bot! # run it!
./bot.pl -c mybot.yaml
 Finally, huge thanks to everyone to helped make the VUC bridging escapade work out – Emil Ivov at Jitsi, James Body, Andy Smith and Giacomo Vacca at Truphone, Anthony Minesalle & Mike Jerris & Brian West at FreeSWITCH for writing freeswitch and mod_verto, Tim Panton for the VUC intro and suggestion of mod_verto, Randy Resnick & Michael Graves at VUC itself, and of course the Matrix team for glueing our side of it together! Looking forward to lots more ambitious cross-protocol gatewaying and federation

in future!</p>

```
                    </div>
```

</html>