automx2

Email client configuration made easy

Ralph Seichter, Patrick Ben Koetter

Version 2021.2, 2021-03-07

Table of Contents

1. About automx2

The project is hosted on GitHub (rseichter/automx2) [https://github.com/rseichter/automx2]. Please use the project's issue tracker in case you experience any problems.

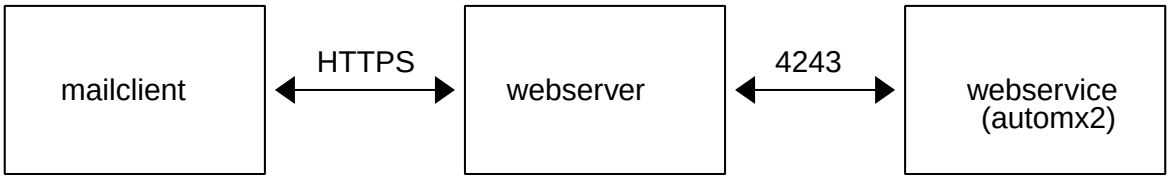
Sponsored by (in alphabetical order) mediaBEAM GmbH [https://www.mediabeam.com/] and sys4 AG [https://sys4.de/].

2. Preface

This document explains how automx2 works, how Automated Mailbox Configuration works and what it takes to install and configure automx2. If you are already familiar with automated mailbox configuration methods you may want skip the following sections and jump right ahead to automx2 Installation and Configuring automx2.

3. How does automx2 work?

automx2 is a webservice. It sits behind a web server, e.g. NGINX, and waits for configuration requests. When a mail client requests configuration it contacts the web server. The web server then acts as a proxy and forwards all requests to the automx2 web service.



4. How does Automated Mailbox Configuration work?

Modern mail clients can look for configuration data when a user begins to create a new account. They will either send the user's mail address to a service and ask the service to reply with configuration that suits the user's profile or they will query the DNS system for advice.

Using a specialized mail account configuration service allows for individualized setups. It also allows to enforce a specific policy, which for example configures the mail client to use a specific authentication mechanism. Quering the DNS for mail service locations allows for generic instructions, but it doesn't give as much control over settings as a specialized service like automx2 will do.

As of today, there are four methods that help configuring a mail account. Three of them – autocon-

fig, autodiscover and mobileconfig – have been developed by vendors to cover their products' specific needs. The fourth is an RFC standard specifying the aformentioned more general DNS SRV Records method.

The vendor specific methods have in common that the mail client seeking configuration needs to send a request, which includes at least the user's mail address, to a configuration service. The service will use the mail address to lookup configuration data and will return that data as response to the client. Format – XML response or file – and complexity differ depending on the method.

automx2 implements everything to configure a mailbox account. It does not implement functionality to e.g. also configure calendar or address book settings.

The following subsections will explain more detailed how the four methods work.

4.1. autoconfig

Autoconfig is a proprietary method developed by the Mozilla foundation. It was designed to configure a mail account within Thunderbird, and other email suites like Evolution and KMail have adopted [https://wiki.mozilla.org/Thunderbird:Autoconfiguration:ConfigFileFormat] the mechanism.

When a user begins to create a new mail account she is asked to enter her realname and mail address, e.g. *alice@example.com*. Thunderbird will then extract the domainpart (*example.com*) from the mail address and build a list of URIs to search for a configuration web service in the following order:

```
https://autoconfig.thunderbird.net/v1.1/example.com
https://autoconfig.example.com/mail/config-v1.1.xml?emailaddress=alice@example.com
https://example.com/.well-known/autoconfig/mail/config-v1.1.xml
http://autoconfig.thunderbird.net/v1.1/example.com
http://autoconfig.example.com/mail/config-v1.1.xml?emailaddress=alice@example.com
http://example.com/.well-known/autoconfig/mail/config-v1.1.xml
```

A configuration service such as automx2 listening on one of the listed URIs will receive the request, process it and respond with a set of configuration instructions.

Thunderbird will use the instructions to automatically fill in the required fields in the account. The only remaining task for the user is to confirm the settings. After that she can immediately start to use her new mail account.

4.2. autodiscover

Autodiscover is a proprietary method developed by Microsoft. It was designed to configure a mail account within Outlook and has expanded to also configure Office 365. Service lookups use the following URLs and, as a fallback option, DNS lookups.

```
https://example.com/autodiscover/autodiscover.xml
https://autodiscover.example.com/autodiscover/autodiscover.xml
http://autodiscover.example.com/autodiscover/autodiscover.xml

dns: autodiscover.example.com
dns: _autodiscover._tcp.example.com
```

All HTTP(S) queries send a POST request and submit XML which contains information about the account that should be configured. The DNS queries search for a `CNAME` resource record first, which is supposed to redirect the mail client to a resource outside of the mailbox owners domain, e.g. *alice@example.com* would be redirected to `service.example-provider.com` for configuration instructions. If the first DNS query fails the client may be redirected to a configuration service using a `SRV` RR like this:

```
_autodiscover._tcp.example.com.  0   443 service.example-provider.com.
```

The `SRV` RR used in the example above would send Alice's client to `service.example-provider.com` and tell it to send the query to the configuration service on port `443`.

4.3. mobileconfig

TODO - Requests and responses use proprietary content types and the PLIST data format.

4.4. DNS SRV Records

```
_imap._tcp.example.com          SRV  10  20  143  mail.example.com.
_imaps._tcp.example.com         SRV  0   1   993  .
_pop3._tcp.example.com          SRV  0   1   110  .
_pop3s._tcp.example.com         SRV  0   1   995  .
_smtp._tcp.example.com.         SRV  0   1   25   .
_submission._tcp.example.com.   SRV  10  20  587  mail.example.com.
```

5. automx2 Installation

automx2 requires Python 3.7 or higher, ideally in the form of a virtual Python environment, to run. Check the python3 version like this:

```
$ python3 --version
Python 3.9.2
```

If you see version 3.6 or lower, you'll need to either change the active Python version for the shell session or edit setupvenv.sh [contrib/setupvenv.sh] after downloading the script.

Don't run as root

If you use a port number greater than 1024 (we suggest 4243), the application does not require super user privileges when running. It also does not need to be installed as root. It is recommended that you create a user account specifically for automx2, but other unprivileged users will do as well.

Prepare the virtual environment for the automx2 web service, adjusting the installation path to your taste (automx2 itself does not care).

```
mkdir -p /srv/web/automx2
cd /srv/web/automx2
```

Download the script that will download and setup your automx2 service:

```
wget -O setupvenv.sh
'https://github.com/rseichter/automx2/raw/master/contrib/setupvenv.sh'
chmod u+x setupvenv.sh
```

Execute the setup script. It will create a Python virtual environment called `venv` in the current directory:

```
./setupvenv.sh
```

Activate the virtual environment and install the latest automx2 release from PyPI. Make sure to pick the correct activation for your shell from the `venv/bin` directory. This is an example for BASH:

```
. venv/bin/activate
pip install automx2
```

Updating automx2

Change to the directory where automx2 has been installed previously. Activate the virtual environment as usual and use pip's `-U` option to update automx2:

```
cd /srv/web/automx2
. venv/bin/activate
pip install -U automx2
```

The next section explains how to configure automx2.

6. Configuring automx2

automx2 uses a file to read runtime instructions from and a database to lookup mail account con-

figuration data.

6.1. Runtime configuration

The configuration file defines automx2 runtime behaviour and it specifies the backend automx2 should read mailbox account configuration data from.

Running without runtime config

If you launch automx2 without a configuration file, it will use internal defaults. These are suitable for testing only. Launched without a config it will use an in-memory SQLite database and all data will be lost once the application terminates.

During startup automx2 searches for runtime configuration instructions in the following locations:

```
AUTOMX2_CONF     ①
~/.automx2.conf  ②
/etc/automx2/automx2.conf
/etc/automx2.conf
```

① If this environment variable exists, it will be used. The value must point to a location where automx2 can read configuration from. If the specified file does not exist, the search will continue.

② If automx2 finds `.automx2.conf` in the home directory of the user that runs automx2, it will be used.

To specify parameters and options automx2 uses an INI style [https://docs.python.org/3.9/library/configparser.html#supported-ini-file-structure] configuration syntax. The example configuration [https://github.com/rseichter/automx2/blob/master/contrib/automx2-sample.conf] that ships with automx2 looks like this:

```
[automx2]
# A typical production setup would use loglevel = WARNING
loglevel = DEBUG
# Echo SQL commands into log? Used for debugging.
db_echo = yes
# In-memory SQLite database
db_uri = sqlite:///:memory:

# SQLite database in a UNIX-like file system
#db_uri = sqlite:////var/lib/automx2/db.sqlite

# MySQL database on a remote server. This example does not use an encrypted
# connection and is therefore *not* recommended for production use.
#db_uri = mysql://username:password@server.example.com/db

# Number of proxy servers between automx2 and the client (default: 0).
# If your logs only show 127.0.0.1 or ::1 as the source IP for incoming
# connections, proxy_count probably needs to be changed.
#proxy_count = 1
```

Place the content of the example configuration into one of the configuration locations automx2 looks for and adapt it to your needs. Then configure the database backend with data that suits your setup, as described below.

6.2. Testing standalone automx2

If you want to verify a vanilla installation of automx2 works, you can populate it with internal test data. Start automx2 as described in section Running automx2 and send the following request to populate your database:

```
curl 'http://127.0.0.1:4243/initdb/'
```

This example assumes you are running automx2 on localhost listening on TCP port 4243, which is the suggested default port.

Once you have populated the database with sample data you can test if automx2 works. Use curl to send an account configuration request for user@example.com:

```
curl 'http://127.0.0.1:4243/mail/config-v1.1.xml?emailaddress=user@example.com'
```

As shown in the example, make sure to use quotes as necessary to prevent your shell from matching/replacing patterns.

6.3. Database configuration

automx2 uses the SQLAlchemy toolkit to access databases. This allows a variety of databases, a.k.a. dialects [https://docs.sqlalchemy.org/en/latest/dialects/], to be used, simply by defining the appropriate connection URL.

API based configuration

I consider adding an API for configuration changes in an upcoming version but have not decided when that might happen. Feel free to contact me if you are interested in sponsoring this feature.

6.3.1. Database support

While you probably already have SQLite support available on your local machine, you may need to install additional Python packages for PostgreSQL, MySQL, etc. Detailed instructions to support a particular database dialect are out of scope for this document. Please search the Internet for detailed instructions on supporting a particular dialect. The SQLAlchemy documentation provides a useful starting point.

While the `contrib` directory contains example database schemas which you can use as a reference, I recommend using the built-in method to create the necessary DB structure from scratch by accessing the `/initdb/` service endpoint. This will also populate the database with some example data.

If you upgrade from an early automx2 release and wish to migrate your existing database, you can use the built-in Alembic support. However, this requires cloning the Git repository, modifying `alembic.ini` and invoking the migration from the command line. It is usually easier to export your existing data, create a fresh DB and import the data.

6.3.2. SQLite

This section demonstrates what you need to do to in order to use SQLite version 3 or higher as a backend database for automx2.

Step 1: Set the database URI in your automx2 configuration. Please note that specifying an absolute path for the database requires a total of four slashes after the schema identifier:

```
[automx2]
db_uri = sqlite:////var/lib/automx2/db.sqlite
```

Step 2: Launch automx2 and access the DB initialisation URL:

```
curl 'http://127.0.0.1:4243/initdb/'
```

The Git repository contains a sqlite-generate.sh helper script which demonstrates how the database can be populated programmatically. You only need to adapt a few settings according to your needs:

```
# User configurable section -- START
PROVIDER_NAME='Example Inc.'
PROVIDER_SHORTNAME='Example'
PROVIDER_ID=100

DOM='example'
TLD='com'
```

The script will print the SQL statements to standard output, which can be piped into `sqlite3` for processing.

```
contrib/sqlite-generate.sh | sqlite3 /var/lib/automx2/db.sqlite
```

This example assumes you have also set `/var/lib/automx2/db.sqlite` as path in your `automx2.conf`.

Placeholders

See Mozilla's placeholder [https://wiki.mozilla.org/Thunderbird:Autoconfiguration:ConfigFileFormat#Placeholders] documentation for further details.

Once you have populated the database automx2 is ready to run.

6.3.3. MySQL

Step 1: Create a database.

```
CREATE DATABASE `automx2` COLLATE 'utf8mb4_general_ci';
```

Step 2: Set the database URI in your automx2 configuration. The following example uses *pymysql* as a DB driver, which is not included in the automx2 distribution.

```
[automx2]
db_uri = mysql+pymysql://user:pass@dbhost/automx2?charset=utf8mb4
```

Step 3: Launch automx2 and access the DB initialisation URL:

```
curl 'http://127.0.0.1:4243/initdb/'
```

6.3.4. PostgreSQL

Step 1: Create a database.

```
CREATE DATABASE automx2 LOCALE 'en_US.utf8';
```

Step 2: Set the database URI in your automx2 configuration. The following example uses *psycopg2-binary* as a DB driver, which is not included in the automx2 distribution.

```
[automx2]
db_uri = postgresql+psycopg2://user:pass@dbhost/automx2
```

Step 3: Launch automx2 and access the DB initialisation URL:

```
curl 'http://127.0.0.1:4243/initdb/'
```

6.4. Running automx2

Running automx2 requires to start automx2 as service and serve its output via a webserver to the public. You should not run automx2 with superuser privileges. Use a dedicated user instead.

The following examples assume you have created a user and group `automx2` and have granted appropriate rights to this user:

- Read permissions for the `automx2.conf` configuration file.
- Read and access permissions for the virtual Python environment.
- Read and access permissions for the SQLite database.

6.4.1. Systemd service

If your system uses systemd you may want to deploy the following automx2.service unit file from the contrib section and place it in `/etc/systemd/system/automx2.service`:

```
[Unit]
After=network.target
Description="automx2 - MUA configuration service"
Documentation=https://github.com/rseichter/automx2/blob/master/doc/automx2.adoc

[Service]
Environment=FLASK_APP=automx2.server:app
Environment=FLASK_CONFIG=production
ExecStart=/srv/www/automx2/bin/flask run --host=127.0.0.1 --port=4243
Restart=always
User=automx2
WorkingDirectory=/var/lib/automx2

[Install]
WantedBy=multi-user.target
```

Once you have installed the service you need to tell systemd to reload its list of available services:

```
sudo systemctl daemon-reload
```

It should now be able to tell you about a service named automx2:

```
sudo systemctl status automx2
● automx2.service - "automx2 - MUA configuration service"
     Loaded: loaded (/etc/systemd/system/automx2.service; disabled; vendor preset:
enabled)
     Active: inactive (dead)
```

Next enable and start automx2 using the following command:

```
sudo systemctl enable automx2 --now
Created symlink /etc/systemd/system/multi-user.target.wants/automx2.service →
/etc/systemd/system/automx2.service.
```

You should see automx2 enabled and running:

```
sudo systemctl status automx2
● automx2.service - "automx2 - MUA configuration service"
     Loaded: loaded (/etc/systemd/system/automx2.service; enabled; vendor preset:
enabled)
     Active: active (running) since Mon 2021-03-01 12:54:31 CET; 19s ago
   Main PID: 126966 (python)
      Tasks: 1 (limit: 4620)
     Memory: 46.1M
     CGroup: /system.slice/automx2.service
             └─126966 /srv/www/automx2/bin/flask run --host=127.0.0.1 --port=4243
[...]
Mar 01 12:54:32 mail python[126966]: Reading /etc/automx2.conf
Mar 01 12:54:32 mail python[126966]: Config.get: loglevel = WARNING
Mar 01 12:54:32 mail python[126966]:  * Running on http://127.0.0.1:4243/ (Press
CTRL+C to quit)
```

You are now ready to start testing automx2, as described in Testing automx2 on command line.

6.4.2. Command line

While logged in as an unprivileged user, change into the installation directory and start the `contrib/flask.sh` launch script:

```
cd /srv/web/automx2
contrib/flask.sh run --host=127.0.0.1 --port=4243
```

Handling terminal output

The flask.sh script will deliberately keep automx2 running in the foreground, and log data will be displayed in the terminal. If you press Ctrl-C or close the shell session, the application will terminate. To run automx2 in the background, you can use a window manager like GNU Screen [https://www.gnu.org/software/screen/] or tmux.

Now that automx2 is up and running, you need to configure the web server proxy that will receive requests from the outside and forwards them to automx2.

6.5. Testing automx2 on command line

Use *curl* to send a request to your local automx2-instance. The following example assumes your service runs on localhost on port 4243. The exact output depends on your database content, but should look similar.

```
curl 'http://127.0.0.1:4243/mail/config-v1.1.xml?emailaddress=user@example.com'
```

```
<clientConfig version="1.1">
    <emailProvider id="automx2-100">
        <identity/>
        <domain>example.com</domain>
        <displayName>Example Inc.</displayName>
        <displayShortName>Example</displayShortName>
        <incomingServer type="imap">
            <hostname>mail.example.com</hostname>
            <port>993</port>
            <socketType>SSL</socketType>
            <username>%EMAILADDRESS%</username>
            <authentication>plain</authentication>
        </incomingServer>
        <incomingServer type="pop3">
            <hostname>mail.example.com</hostname>
            <port>110</port>
            <socketType>STARTTLS</socketType>
            <username>%EMAILADDRESS%</username>
            <authentication>plain</authentication>
        </incomingServer>
        <outgoingServer type="smtp">
            <hostname>mail.example.com</hostname>
            <port>587</port>
            <socketType>STARTTLS</socketType>
            <username>%EMAILADDRESS%</username>
            <authentication>plain</authentication>
        </outgoingServer>
        <!-- ... -->
    </emailProvider>
</clientConfig>
```

Having verified that automx2 returns configuration data, you should make the service available using a web server as a proxy.

6.6. Configuring a web server

While it is technically possible to run automx2 without a web server sitting in front of it, we don't recommend doing that in a production environment. A web server can provide features automx2 was designed not to have. Features such as transport layer encryption for HTTPS or, for example, the capability to rate-limit clients are handled very well by full-fledged web servers working as reverse proxies. It would be a waste to re-implement all this in a web service.

This section will explain how to configure a web server as a reverse proxy in front of automx2. Before you set up the proxy you need to tell automx2 it operates behind one. Add the `proxy_count` parameter to your automx2 configuration file or uncomment the parameter if it is already there:

```
[automx2]
# A typical production setup would use loglevel = WARNING
loglevel = WARNING

# Disable SQL command echo.  ①
db_echo = no

# SQLite database in a UNIX-like file system
db_uri = sqlite:////var/lib/automx2/db.sqlite

# Number of proxy servers between automx2 and the client (default: 0).
# If your logs only show 127.0.0.1 or ::1 as the source IP for incoming
# connections, proxy_count probably needs to be changed.  ②
proxy_count = 1
```

① You might want to turn off echoing SQL commands to the log once you have verified automx2 works as expected.

② Set the number to reflect the number of proxies chained in front of automx2, i.e. the number of "proxy hops" a client's request must pass before it reaches automx2.

6.6.1. NGINX

The following example defines a HTTP server, which will listen for requests to both *autoconfig.example.com* and *autodiscover.example.com*. All requests will be forwarded to automx2, which listens on 127.0.0.1 on port 4243 in this example. Requests to `/initdb` are restricted to clients from 127.0.0.1 only. The `proxy_set_header` directives will cause NGINX to pass relevant data about incoming requests' origins.

```
# NGINX example configuration snippet to forward incoming requests to automx2.
# vim:ts=4:sw=4:et:ft=nginx

http {
    server {
        listen *:80;
        listen [::]:80;
        server_name autoconfig.example.com autodiscover.example.com;
        location /initdb {
            # Limit access to clients connecting from localhost
            allow 127.0.0.1;
            deny all;
        }
        location / {
            # Forward all traffic to local automx2 service
            proxy_pass http://127.0.0.1:4243/;
            proxy_set_header Host $host;
            # Set config parameter proxy_count=1 to have automx2 process these headers
            proxy_set_header X-Forwarded-Proto http;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

6.6.2. Apache

The following example shows an Apache configuration similar to the one above. `ProxyPreserveHost` directives will cause apache to pass relevant data about incoming requests' origins.

```
# Apache 2.4 example configuration snippet to forward incoming requests to automx2.
# vim:ts=4:sw=4:et:ft=apache

<VirtualHost *:80>
    ServerName autoconfig.example.com
    ServerAlias autodiscover.example.com
    ProxyPreserveHost On
    ProxyPass "/" "http://127.0.0.1:4243/"
    ProxyPassReverse "/" "http://127.0.0.1:4243/"
    <Location /initdb>
        # Limit access to clients connecting from localhost
        Order Deny,Allow
        Deny from all
        Allow from 127.0.0.1
    </Location>
</VirtualHost>
```