



You:

I have a server with a static ip associated with a domain (plan6.com) that has successfully sent an email with its own mail server in the past. It has a txt entry in its DNS v=spf1 ip4:74.208.253.216 -all. Port 25 is not blocked or firewalled. It's Domain names has a correct reverse lookup set up for it. In short, with the exception of a mail server installation, the server is prepared to become an email server. I want to use this email server to send emails from a Django backend on a different server (backend.plan6.com). But before doing that I want to test it with this code running on my peronal computer (separate from the server):

```
import smtplib from email.message import EmailMessage def send_email(): # Define the email parameters smtp_server = "plan6.com" smtp_port = 25 sender_email = "billing@plan6.com" sender_password = "-----redacted-----" recipient_email = "twcookson@gmail.com" subject = "Test Email 2" body = "This is a test email sent from Python." # Create the email msg = EmailMessage() msg.set_content(body) msg["Subject"] = subject msg["From"] = sender_email msg["To"] = recipient_email # Send the email with smtplib.SMTP(smtp_server, smtp_port) as server: # Uncomment the next line if you're using SSL # server.starttls() server.login(sender_email, sender_password) server.send_message(msg) print("Email sent successfully!") send_email()
```

Given this information, what's the easiest mail server program I can install on my server to achieve these (sending of mail from a remote python request). What are my options and what are the pros and cons?



ChatGPT:

For setting up a simple mail server to achieve sending mail from a remote Python request, there are a few common mail server programs you can consider:

#### 1. Postfix:

- **Pros:**
  - Popular and widely used.
  - Reliable and robust.
  - Good integration with various other software.
  - Has a lot of documentation available.
  - Suitable for both small and large scale setups.
- **Cons:**
  - Initial setup can be a bit complex for beginners.
  - Requires proper security configuration to prevent misuse.

#### 2. Exim:

- **Pros:**
  - Also widely used, especially on some cPanel servers.
  - Configurable and flexible.
  - Reliable for sending emails.
- **Cons:**
  - Like Postfix, the initial setup can be confusing for beginners.

#### 3. Sendmail:

- **Pros:**
  - One of the oldest and most established mail servers.
  - Supported on many systems by default.
- **Cons:**
  - Configuration can be daunting.
  - Not as performant as some of the newer alternatives.
  - Has a history of security vulnerabilities.

#### 4. OpenSMTPD:

- **Pros:**
  - Simplified configuration compared to Postfix and Sendmail.
  - Secure and built with modern standards in mind.
- **Cons:**
  - Not as feature-rich as Postfix or Exim.
  - Might not have as many online resources or community support as the more popular solutions.

Given your requirements, I would recommend **Postfix**. It is modern, secure, and versatile. While the initial setup may seem a bit daunting, there are many tutorials and resources available that can guide you through the process. Postfix can also scale as your needs grow, which can be beneficial in the future.

To install and set up Postfix:

##### 1. Install Postfix: For Debian/Ubuntu systems:

```
bash
sudo apt update
sudo apt install postfix
```

For Red Hat/CentOS systems:

```
bash
```

```
sudo yum install postfix
sudo systemctl enable postfix
```

2. **Configure Postfix:** After installing, you'll be prompted to select some basic configuration options. For your use case, you can select "Internet Site" and enter the domain name when prompted.
3. Make sure to properly secure your Postfix installation to prevent it from being used for spam. This would involve settings such as:
  - Disabling open relay.
  - Enabling SASL authentication.
  - Implementing SPF, DKIM, and DMARC records for better mail deliverability.

Lastly, ensure your server's firewall is configured to allow traffic on port 25 (for SMTP).

Your provided Python code will work without issues if you properly configure the mail server. Ensure that you replace "-----redacted-----" with the actual password for "[billing@plan6.com](mailto:billing@plan6.com)" and also set up this email account on your mail server.



Tell me how to install and setup OpenSMTPD on a Ubuntu LTS server where it will be use to send mail like this code run from another computer: 

```
import smtplib from email.message import EmailMessage def send_email(): # Define the email parameters smtp_server = "plan6.com" smtp_port = 25 sender_email = "billing@plan6.com" sender_password = "-----redacted-----" recipient_email = "twcookson@gmail.com" subject = "Test Email 2" body = "This is a test email sent from Python." # Create the email msg = EmailMessage() msg.set_content(body) msg["Subject"] = subject msg["From"] = sender_email msg["To"] = recipient_email # Send the email with smtplib.SMTP(smtp_server, smtp_port) as server: # Uncomment the next line if you're using SSL # server.starttls() server.login(sender_email, sender_password) server.send_message(msg) print("Email sent successfully!") send_email()
```



Certainly! OpenSMTPD offers a simplified configuration compared to other mail servers, which makes it an attractive choice for your use case. Here's a step-by-step guide on setting up OpenSMTPD on an Ubuntu LTS server:

### 1. Installing OpenSMTPD:

```
bash
sudo apt update
sudo apt install opensmtpd
```

### 2. Basic Configuration:

Edit the OpenSMTPD configuration file:

```
bash
sudo nano /etc/smtpd/smtpd.conf
```

A basic configuration to allow sending email would look like this:

```
perl
listen on localhost

# Authentication table
table secrets file:/etc/smtpd/secrets

# Mail authentication
action "local_mail" mbox alias <aliases>
action "outgoing_mail" relay host smtp+auth://billing@plan6.com:-----redacted-----@localhost auth <secrets>

# Match rules
match for local action "local_mail"
match for any action "outgoing_mail"
```

Make sure you replace "-----redacted-----" with your actual password.

### 3. Setup Email Authentication:

Now, we set up the authentication table:

```
bash
```

```
echo "billing@plan6.com:-----redacted-----" | sudo tee /etc/smtpd/secrets
```

Again, replace "-----redacted-----" with the actual password.

Set the appropriate permissions:

```
bash
sudo chown _smtpd:_smtpq /etc/smtpd/secrets
sudo chmod 640 /etc/smtpd/secrets
```

#### 4. Starting and Enabling OpenSMTPD:

Start OpenSMTPD:

```
bash
sudo systemctl start opensmtpd
```

Enable it to start on boot:

```
bash
sudo systemctl enable opensmtpd
```

#### 5. Open Ports in Firewall:

If you're using UFW (Uncomplicated Firewall) or another firewall solution, make sure to allow port 25:

```
bash
sudo ufw allow 25/tcp
```

#### Notes:

1. This setup is quite basic and intended for sending emails only. It's important to understand that, if misconfigured, mail servers can be exploited to send spam. Ensure you keep your server and software up-to-date, and monitor for suspicious activity.
2. Ensure that the domain's DNS settings (SPF, DKIM, etc.) are correctly configured to improve email deliverability.
3. Depending on the email volume, it might be good to consider rate limiting and other security measures.
4. Ensure you back up configurations before making changes, and test thoroughly before deploying in a production environment.

Your Python code should work with this setup, provided you're using the right credentials and DNS settings.

-----



You:

I found this file: /etc/smtpd.conf with this text # \$OpenBSD: smtpd.conf,v 1.10 2018/05/24 11:40:17 gilles Exp \$ # This is the smtpd server system-wide configuration file. # See smtpd.conf(5) for more information. table aliases file:/etc/aliases # To accept external mail, replace with: listen on all # listen on localhost action "local" maildir alias <aliases> action "relay" relay # Uncomment the following to accept external mail for domain "example.org" # # match from any for domain "example.org" action "local" match for local action "local" match from local for any action "relay" Is that the file I should be editing instead of sudo nano /etc/smtpd/smtpd.conf

-----



ChatGPT:

Yes, the correct file is /etc/smtpd.conf on Ubuntu. My apologies for the oversight.

Given the contents of your /etc/smtpd.conf, let's make the necessary modifications to configure it for sending emails as per your requirements.

Here's how you can modify the provided configuration:

1. Open the configuration file:

```
bash
sudo nano /etc/smtpd.conf
```

2. Update the configuration to look like the following:

```
bash
table aliases file:/etc/aliases
table secrets file:/etc/smtpd/secrets
```

```
# Listen on all available network interfaces, you can change it to "listen on localhost" if you only want to allow
listen on all

action "local" maildir alias <aliases>
action "relay" relay
action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets>

match for local action "local"
match from local for any action "relay"
match auth from any for any action "authenticated"
```

3. As previously mentioned, you'll need to create and set up the authentication table:

```
bash
echo "billing@plan6.com:-----redacted-----" | sudo tee /etc/smtpd/secrets
```

Then set the appropriate permissions:

```
bash
sudo chown _smtpd:_smtpq /etc/smtpd/secrets
sudo chmod 640 /etc/smtpd/secrets
```

4. Finally, restart the OpenSMTPD service to apply the changes:

```
bash
sudo systemctl restart opensmtpd
```

The configuration will now accept authenticated SMTP connections and allow sending emails. You should be able to use your Python script with the email server.



:/etc/smtpd/secrets There's no smtpd folder and no secrets file in the root of /etc/



I apologize for the oversight. The path I mentioned is not the standard path for OpenSMTPD on Ubuntu. Let's correct that.

Instead of creating a secrets file in a non-existent directory, you can place it directly under /etc/.

#### 1. Create and Set Up the Authentication Table:

```
bash
echo "billing@plan6.com:-----redacted-----" | sudo tee /etc/smtpd_secrets
```

Replace "-----redacted-----" with your actual password.

Then, set the appropriate permissions:

```
bash
sudo chown root:_smtpd /etc/smtpd_secrets
sudo chmod 640 /etc/smtpd_secrets
```

#### 2. Modify the /etc/smtpd.conf Configuration:

You'll need to reflect the correct path to the smtpd\_secrets file in your /etc/smtpd.conf configuration.

Open the configuration file:

```
bash
sudo nano /etc/smtpd.conf
```

Update the table secrets line to:

```
javascript
```

```
table secrets file:/etc/smtpd_secrets
```

After making these changes, proceed to restart the OpenSMTPD service:

```
bash
sudo systemctl restart opensmtpd
```

With these steps, the authentication should be set up correctly.



```
tom@plan6:/etc$ sudo chown root:_smtpd /etc/smtpd_secrets
chown: invalid group: 'root:_smtpd'
```



I apologize for the confusion. The group `_smtpd` is not standard on Ubuntu. Instead, the typical group associated with mail services is `mail`.

Try the following:

```
bash
sudo chown root:mail /etc/smtpd_secrets
```

Then, set the appropriate permissions:

```
bash
sudo chmod 640 /etc/smtpd_secrets
```

This should set the owner of the file to the `root` user and the group to `mail`, while ensuring that only the owner can write to it but members of the `mail` group can read it.



```
What do you make of this: tom@plan6:/etc$ sudo systemctl status opensmtpd
● opensmtpd.service - OpenSMTPD SMTP server
   Loaded: loaded (/lib/systemd/system/opensmtpd.service; enabled; vendor preset: enabled)
   Active: failed (Result: exit-code) since Sun 2023-09-03 18:59:57 UTC; 1min 22s ago
   Docs: man:smtpd(8)
   Process: 5319 ExecStart=/usr/sbin/smtpd (code=exited, status=0/SUCCESS)
   Main PID: 5320 (code=exited, status=1/FAILURE)
   CPU: 57ms
   Sep 03 18:59:57 plan6.com systemd[1]: Starting OpenSMTPD SMTP server...
   Sep 03 18:59:57 plan6.com smtpd[5319]: info: OpenSMTPD 6.8.0p2 starting
   Sep 03 18:59:57 plan6.com systemd[1]: Started OpenSMTPD SMTP server.
   Sep 03 18:59:57 plan6.com smtpd[5324]: pony express: smtpd: bind: Cannot assign requested address
   Sep 03 18:59:57 plan6.com systemd[1]: opensmtpd.service: Main process exited, code=exited, status=1/FAILURE
   Sep 03 18:59:57 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'.
```



The log you provided gives some indication of what might be wrong. This line stands out:

```
bash
pony express: smtpd: bind: Cannot assign requested address
```

This suggests that OpenSMTPD is trying to bind to an address (or port) that is either already in use or not available.

Let's try troubleshooting this:

#### 1. Check for Port Conflicts:

Ensure that no other service is using port 25, which is the default port for SMTP.

```
bash
sudo netstat -tuln | grep ":25"
```

If you see any output, it means something is already using port 25.

## 2. Listen on Localhost:

It's possible that `listen on all` is trying to bind to an IP address that is not assigned to any network interface on your server. For testing purposes, change `listen on all` to `listen on localhost` in your `/etc/smtpd.conf` file.

After making this change, try to restart OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

If it starts successfully, then the issue is likely related to the server trying to bind to a non-local or unavailable IP address.

## 3. Check Configuration:

OpenSMTPD provides a useful tool to check the configuration file for syntax and potential issues.

```
bash
smtpd -n
```

This command should tell you if there are any issues in your configuration file.

## 4. Review Logs for Additional Details:

Review the detailed logs of OpenSMTPD to see if there's more information on why it might be failing:

```
bash
sudo journalctl -u opensmtpd
```

This will display logs specifically related to OpenSMTPD.

By working through these steps, you should be able to identify the problem or at least gather more information that will help in pinpointing the issue.



You:

```
tom@plan6:/etc$ sudo netstat -tln | grep ":25" udp6 0 0 fe80::250:56ff:fe1c:546 :::* tom@plan6:/etc$
```



ChatGPT:

The output shows that there's a service listening on an IPv6 address (`fe80::250:56ff:fe1c:546`) on port 25, but it's UDP, not TCP. SMTP runs over TCP, so this isn't conflicting with OpenSMTPD.

Since that doesn't appear to be the issue, let's proceed:

1. As previously suggested, try modifying the `/etc/smtpd.conf` file to listen only on `localhost`:

```
arduino
listen on localhost
```

Then restart OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

2. If the error persists, run the configuration check for OpenSMTPD:

```
bash
smtpd -n
```

3. If no errors are shown, check the detailed logs:

```
bash
sudo journalctl -u opensmtpd
```

This might give more information about why the service isn't starting correctly.

If after trying these steps, you're still facing issues, you might want to consider looking into other configurations on the server or even alternative SMTP solutions depending on your comfort level and requirements.



You:

-- Journal begins at Thu 2023-08-10 17:21:50 UTC, ends at Sun 2023-09-03 19:04:13 UTC. -- Sep 03 18:48:36 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 18:48:36 plan6.com smtpd[5197]: info: OpenSMTPD 6.8.0p2 starting Sep 03 18:48:36 plan6.com systemd[1]: Started OpenSMTPD SMTP server. Sep 03 18:59:57 plan6.com systemd[1]: Stopping OpenSMTPD SMTP server... Sep 03 18:59:57 plan6.com smtpd[5207]: Exiting Sep 03 18:59:57 plan6.com systemd[1]: opensmtpd.service: Succeeded. Sep 03 18:59:57 plan6.com systemd[1]: Stopped OpenSMTPD SMTP server. Sep 03 18:59:57 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 18:59:57 plan6.com smtpd[5319]: info: OpenSMTPD 6.8.0p2 starting Sep 03 18:59:57 plan6.com systemd[1]: Started OpenSMTPD SMTP server. Sep 03 18:59:57 plan6.com smtpd[5324]: pony express: smtpd: bind: Cannot assign requested address Sep 03 18:59:57 plan6.com systemd[1]: opensmtpd.service: Main process exited, code=exited, status=1/FAILURE Sep 03 18:59:57 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:03:51 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:03:51 plan6.com smtpd[5350]: info: OpenSMTPD 6.8.0p2 starting Sep 03 19:03:51 plan6.com systemd[1]: Started OpenSMTPD SMTP server.



ChatGPT:

The error "pony express: smtpd: bind: Cannot assign requested address" is the primary issue. It means that OpenSMTPD is trying to bind to an IP address (or port) that isn't available on the system. Let's try to address it.

### 1. Specify IP Address to Listen On:

Instead of `listen` on `localhost`, try specifying the IP address directly. If your server's IP address is, for example, `192.168.1.100`, modify the `/etc/smtpd.conf` to:

```
arduino
listen on 192.168.1.100 port 25
```

Then restart the service.

### 2. Check IPv6 Configuration:

If your system is trying to use IPv6, and you don't need it, you could disable IPv6. This can be done by editing the `/etc/sysctl.conf` file and adding:

```
arduino
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Then run:

```
bash
sudo sysctl -p
```

Restart OpenSMTPD afterwards.

### 3. Upgrade OpenSMTPD:

Though it's less likely, software bugs or compatibility issues can also lead to binding errors. Ensure you're running the latest version available for Ubuntu. Update your system packages and see if there's an update for OpenSMTPD.

```
bash
sudo apt update
sudo apt upgrade
```

### 4. Check for Multiple Network Interfaces:

If your server has multiple network interfaces, it's possible that there's confusion on which one to bind to. You can list your network interfaces with:

```
bash
ip a
```

If there are multiple IPs, specify one explicitly in the `smtpd.conf` file as mentioned in step 1.

### 5. Reconfigure OpenSMTPD:

If you've tried the above solutions and are still facing issues, you might want to reinstall and reconfigure OpenSMTPD. Sometimes, starting with a clean configuration can help in pinpointing issues.

Lastly, if all these steps don't solve the issue, there could be other underlying server configurations or environmental factors that are causing the problem. It might be a good idea to consult with the server hosting provider or review more detailed logs and configurations.



You:

It won't even start after entering the ip6 code, and 192.168.1.100 port 25 didn't change anything: Sep 03 19:05:37 plan6.com systemd[1]: opensmtpd.service: Succeeded. Sep 03 19:05:37 plan6.com systemd[1]: Stopped OpenSMTPD SMTP server. Sep 03 19:05:37 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:05:37 plan6.com smtpd[5371]: info: OpenSMTPD 6.8.0p2 starting Sep 03 19:05:37 plan6.com systemd[1]: Started OpenSMTPD SMTP server. Sep 03 19:05:37 plan6.com smtpd[5376]: pony express: smtpd: bind: Cannot assign requested address Sep 03 19:05:37 plan6.com systemd[1]: opensmtpd.service: Main process exited, code=exited, status=1/FAILURE Sep 03 19:05:37 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:06:24 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:06:24 plan6.com smtpd[5390]: /etc/smtpd.conf:26: syntax error Sep 03 19:06:24 plan6.com smtpd[5390]: /etc/smtpd.conf:27: syntax error Sep 03 19:06:24 plan6.com smtpd[5390]: /etc/smtpd.conf:28: syntax error Sep 03 19:06:24 plan6.com systemd[1]: opensmtpd.service: Control process exited, code=exited, status=1/FAILURE Sep 03 19:06:24 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:06:24 plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server. Sep 03 19:06:33 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:06:33 plan6.com smtpd[5394]: /etc/smtpd.conf:26: syntax error Sep 03 19:06:33 plan6.com smtpd[5394]: /etc/smtpd.conf:27: syntax error Sep 03 19:06:33 plan6.com smtpd[5394]: /etc/smtpd.conf:28: syntax error Sep 03 19:06:33 plan6.com systemd[1]: opensmtpd.service: Control process exited, code=exited, status=1/FAILURE Sep 03 19:06:33 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:06:33 plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server.



ChatGPT:

Thank you for the update. Let's address the issues step by step:

### 1. Syntax Errors in smtpd.conf:

The recent logs indicate that there are syntax errors on lines 26, 27, and 28 of the `/etc/smtpd.conf` file. We need to inspect and correct these first.

Use a text editor to open the configuration file:

```
bash
sudo nano /etc/smtpd.conf
```

Examine lines 26-28 and correct any syntax issues. If you're unsure about the changes, you can paste the contents here, and I'll help you correct them.

### 2. IPv6 Configuration:

If you've added lines related to disabling IPv6 in the `/etc/sysctl.conf` file, but it caused OpenSMTPD not to start, I recommend commenting them out for now.

Open `/etc/sysctl.conf`:

```
bash
sudo nano /etc/sysctl.conf
```

Add a `#` at the beginning of the lines related to disabling IPv6:

```
arduino
#net.ipv6.conf.all.disable_ipv6 = 1
#net.ipv6.conf.default.disable_ipv6 = 1
#net.ipv6.conf.lo.disable_ipv6 = 1
```

Save and close the file.

To apply changes:

```
bash
sudo sysctl -p
```

### 3. IP Address in smtpd.conf:

If you're unsure of the server's IP address to set for the `listen` on directive, let's determine it. Use the following command:

```
bash
ip a | grep inet
```



You should see a list of IP addresses. One of them, typically not starting with 127.0.x.x (which is localhost), is your server's IP. Use this IP in the `listen` on directive.

Once you've made the necessary corrections to the configuration files, attempt to restart OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

Check the status again:

```
bash
sudo systemctl status opensmtpd
```

Let's address the issues incrementally to avoid compounding problems.



You:

```
tom@plan6:/etc$ ip a | grep inet inet 127.0.0.1/8 scope host lo inet6 ::1/128 scope host inet 74.208.253.216/32 brd 74.208.253.216 scope global dynamic ens192 inet6 fe80::250:56ff:fe1c:aa18/64 scope link
```



ChatGPT:

Okay, your server's public IPv4 address is 74.208.253.216.

Let's proceed with the following steps:

### 1. Editing `smtpd.conf`:

Open the `/etc/smtpd.conf` file:

```
bash
sudo nano /etc/smtpd.conf
```

Update the `listen` on directive to use your server's IP address:

```
plaintext
listen on 74.208.253.216
```

Ensure you address the syntax errors on lines 26, 27, and 28 as previously mentioned.

### 2. Restart the Service:

Once the configuration is edited and saved, restart the OpenSMTPD service:

```
bash
sudo systemctl restart opensmtpd
```

Check the status:

```
bash
sudo systemctl status opensmtpd
```

If there are any more issues, the status command should provide more detailed error messages.

### 3. Firewall Considerations:

Ensure your firewall allows incoming and outgoing connections on port 25.

If you're using `ufw`, the command to allow SMTP is:

```
bash
sudo ufw allow 25/tcp
```

And then reload the firewall:

```
bash
```

```
sudo ufw reload
```

This should help you get OpenSMTPD up and running with the current configuration. If you still face issues, let's see the output of the status command or any relevant logs.

 You:

```
Good job. Looks promising: om@plan6:/etc$ sudo nano /etc/smtpd.conf tom@plan6:/etc$ sudo systemctl restart opensmtpd tom@plan6:/etc$ sudo
systemctl status opensmtpd ● opensmtpd.service - OpenSMTPD SMTP server Loaded: loaded (/lib/systemd/system/opensmtpd.service; enabled; vendor
preset: Active: active (running) since Sun 2023-09-03 19:13:47 UTC; 5s ago Docs: man:smtpd(8) Process: 5555 ExecStart=/usr/sbin/smtpd
(code=exited, status=0/SUCCESS) Main PID: 5556 (smtpd) Tasks: 7 (limit: 465) Memory: 10.0M CPU: 62ms CGroup: /system.slice/opensmtpd.service
├─5556 /usr/sbin/smtpd ──5557 smtpd: klondike ──5558 smtpd: control ──5559 smtpd: lookup ──5560 smtpd: pony express ──5561 smtpd: queue
└─5562 smtpd: scheduler Sep 03 19:13:47 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:13:47 plan6.com smtpd[5555]: info:
OpenSMTPD 6.8.0p2 starting Sep 03 19:13:47 plan6.com systemd[1]: Started OpenSMTPD SMTP server.
```

 ChatGPT:

That's excellent news! The OpenSMTPD service is up and running as indicated by the `active (running)` status.

Now you can proceed to use your Python script to send emails through the server. Just make sure you've configured authentication as necessary and the firewall isn't blocking the necessary ports.

To test if everything is set up properly:

1. **Test Locally:** You can attempt to send an email from the server itself first to ensure that there are no configuration issues.
2. **Test Remotely:** Try sending emails from a different computer or network, ensuring your firewall allows incoming connections on port 25 from external sources.

Always monitor logs and error messages, as they can provide vital insights if anything doesn't work as expected.

 You:

Cool lets test locally

 ChatGPT:

Certainly! Testing locally is a great way to confirm the basic functionality of the SMTP server. Here's how you can do a basic local test:

1. **Install the swaks utility:** swaks (Swiss Army Knife SMTP) is a command-line tool that can be used to test SMTP servers. It's simple and versatile, making it great for this purpose.

```
bash
sudo apt-get install swaks
```

2. **Use swaks to send an email:**

```
bash
swaks --to recipient@example.com --from sender@yourdomain.com --server localhost
```

Replace `recipient@example.com` with a valid email address where you can check the email (e.g., your personal email) and `sender@yourdomain.com` with an address from your domain.

This command will attempt to send a test email to the specified recipient through the SMTP server running on `localhost`.

3. **Check the recipient's inbox:**

- See if the test email arrived.
- Check for any delays or issues.
- Examine the email's headers and content for any anomalies.

4. **Review the logs:**

It's a good idea to look at the SMTP server logs to see if there were any issues or to understand more about the process.

For OpenSMTPD, the logs are typically available through the system's journal:

```
bash
sudo journalctl -u opensmtpd
```

Look for any error or warning messages related to the test you just conducted.

By conducting this local test, you'll be able to determine if the server is capable of sending emails on its own. If this works, the next step would be to test remotely and also to ensure that emails are not being flagged as spam by recipient email servers.



You:

```
om@plan6:/etc$ swaks --to twcookson@gmail.com --from billing@plan6.com --server localhost === Trying localhost:25... *** Error connecting to
localhost:25: *** IO::Socket::INET6: connect: Connection refused Do i need my password in there somewhere for billing@plan6.com?
```



ChatGPT:

The error message you received (Connection refused) indicates that swaks couldn't connect to the SMTP server on localhost:25. This is not related to authentication but rather about the connectivity to the SMTP service itself.

Here are a few things to check:

### 1. Ensure OpenSMTPD is running and listening on port 25:

```
bash
sudo netstat -tuln | grep ":25"
```

This should show that something (hopefully OpenSMTPD) is listening on port 25.

### 2. Check OpenSMTPD Configuration:

Ensure that the configuration in /etc/smtpd.conf is set to listen on localhost (127.0.0.1) port 25. The relevant line should look something like:

```
bash
listen on localhost port 25
```

If it's not there, add it, and then restart OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

### 3. Try using swaks again:

```
bash
swaks --to twcookson@gmail.com --from billing@plan6.com --server localhost
```

If you've set up OpenSMTPD to require authentication for sending, you'll need to provide the credentials in the swaks command using the --auth-user and --auth-password options:

```
bash
swaks --to twcookson@gmail.com --from billing@plan6.com --server localhost --auth LOGIN --auth-user billing@plan6
```

Note: Please ensure you're using this in a safe environment as the password is entered in plaintext in the command line, which can be seen in your command history.



You:

```
tom@plan6:/etc$ swaks --to twcookson@gmail.com --from billing@plan6.com --server 74.208.253.216 === Trying 74.208.253.216:25... === Connected to
74.208.253.216. <- 220 plan6.com ESMTP OpenSMTPD -> EHLO plan6.com <- 250-plan6.com Hello plan6.com [74.208.253.216], pleased to meet you <-
250-8BITMIME <- 250-ENHANCEDSTATUSCODES <- 250-SIZE 36700160 <- 250-DSN <- 250 HELP -> MAIL FROM:<billing@plan6.com> <- 250 2.0.0
Ok -> RCPT TO:<twcookson@gmail.com> <- 250 2.1.5 Destination address valid: Recipient ok -> DATA <- 354 Enter mail, end with "." on a line by itself ->
Date: Sun, 03 Sep 2023 19:28:48 +0000 -> To: twcookson@gmail.com -> From: billing@plan6.com -> Subject: test Sun, 03 Sep 2023 19:28:48 +0000 ->
```

Message-Id: <20230903192848.005762@plan6.com> -> X-Mailer: swaks v20201014.0 jetmore.org/john/code/swaks/ -> -> This is a test mailing -> -> .  
<- 250 2.0.0 7636fa0a Message accepted for delivery -> QUIT <- 221 2.0.0 Bye === Connection closed with remote host. tom@plan6:/etc\$



Great! The test email was successfully accepted for delivery by your OpenSMTPD server.

The logs show a successful sequence:

1. Connected to the server.
2. Exchanged greetings.
3. Set the sender (MAIL FROM) and it was accepted.
4. Set the recipient (RCPT TO) and it was accepted.
5. Sent the data of the email.
6. The server responded that the message was accepted for delivery.

The next steps would be:

1. **Check the Gmail Inbox:** Ensure the email was delivered to the target Gmail account ([twcookson@gmail.com](mailto:twcookson@gmail.com)). If it's not in the primary inbox, be sure to also check the spam/junk folder.
2. **Logs:** If there are any issues with delivery, you may want to check the mail logs on your server to see if there are any messages about why the email wasn't delivered. Typically, mail logs are found in `/var/log/mail.log` or similar.
3. **External Tools:** You might also want to check the reputation of your IP address to ensure it's not on any blacklists. There are several online tools available that allow you to check if an IP address is blacklisted.

Remember, sending emails from a new server or IP can sometimes get them marked as spam by big email providers (like Gmail). Make sure you have SPF, DKIM, and DMARC properly set up for better deliverability.

But overall, from the output you've provided, things look successful on the SMTP side!



You:

I checked and it worked when I sent the email from the server. Then I used the same commands to send the email remotely:  
tom@laptoplin:~/ParkEzAI/backend\$ swaks --to twcookson@gmail.com --from billing@plan6.com --server 74.208.253.216 === Trying  
74.208.253.216:25... === Connected to 74.208.253.216. <- 220 plan6.com ESMTP OpenSMTPD -> EHLO laptoplin <- 250-plan6.com Hello laptoplin  
[47.17.154.49], pleased to meet you <- 250-8BITMIME <- 250-ENHANCEDSTATUSCODES <- 250-SIZE 36700160 <- 250-DSN <- 250 HELP -> MAIL  
FROM:<billing@plan6.com> <- 250 2.0.0 Ok -> RCPT TO:<twcookson@gmail.com> < \*\* 550 Invalid recipient: <twcookson@gmail.com> -> QUIT <- 221  
2.0.0 Bye === Connection closed with remote host. tom@laptoplin:~/ParkEzAI/backend\$



The key here is the response:

```
graphql
< ** 550 Invalid recipient: <twcookson@gmail.com>
```

This suggests that your OpenSMTPD server is rejecting external mail relay. In the SMTP world, "relaying" means accepting mail from an external source and sending it to another external destination. To prevent spam, many SMTP servers disallow open relaying by default. This is because, without restrictions, anyone could use your server to send emails, which could quickly get your server blacklisted or abused for spam.

Here's what you can do:

#### 1. Allow Relaying for Authenticated Users:

- If OpenSMTPD supports SMTP authentication, you can allow relaying only for authenticated users. This means that when you send an email through your SMTP server from an external source, you would provide a username and password to authenticate.
- To use this method, you would configure OpenSMTPD to require SMTP authentication and modify your swaks command to provide authentication credentials.

#### 2. Limit Relay by IP:

- You can configure OpenSMTPD to allow relaying from specific IP addresses or IP ranges without authentication. If you know the IPs that will be sending mail through your server, you can whitelist just those. However, this approach can be less secure, especially if the IP address is dynamic.

#### 3. Check SMTPD Configuration:

- Open and inspect `/etc/smtpd.conf` to check if there are any restrictions set for relaying. Look for rules that might be disallowing mail relay and adjust accordingly.

#### 4. Avoid Open Relays:

- It's crucial to avoid setting up an "open relay," which would allow anyone on the internet to send emails through your SMTP server. This can lead to your server being used for spam, which can get your IP blacklisted.

If you're open to sharing your OpenSMTPD configuration (while redacting any sensitive information), I can provide more specific guidance.



Okay, I have a password to authenticate and swaks installed on an external machine to test it. server is plan6.com / 72.208.253.216 password is fakepassword email is billing@plan6.com



Great! Here's how you can set up SMTP authentication with OpenSMTPD:

##### 1. Configuring OpenSMTPD to Support Authentication:

Open `/etc/smtpd.conf` and ensure you have the following lines:

```
typescript
listen on 0.0.0.0 port 25
listen on :: port 25

table credentials file:/etc/mail/credentials
accept from any for domain "plan6.com" relay
accept for local deliver to mbox
accept for any relay via smtp+auth://[127.0.0.1] auth <credentials>
```

This configuration does the following:

- It sets OpenSMTPD to listen on all available IP addresses on port 25.
- Defines a table for credentials.
- Accepts emails destined for plan6.com and relays them.
- Accepts emails for local delivery.
- Accepts emails to relay to other domains, but only if authenticated.

##### 2. Setting Up Authentication Credentials:

Create (or edit) the `/etc/mail/credentials` file to have the following:

```
css
billing@plan6.com fakepassword
```

This sets up the billing@plan6.com email address with the password fakepassword for SMTP authentication.

##### 3. Testing from an External Machine:

Use swaks on an external machine to test sending with authentication:

```
scss
swaks --to twcookson@gmail.com --from billing@plan6.com --server 74.208.253.216 --auth-user billing@plan6.com
```

This will use SMTP authentication when sending the email.

Make sure to restart OpenSMTPD after making these changes:

```
sudo systemctl restart opensmtpd
```

A couple of notes:

- SMTP authentication typically involves sending the password in plaintext or using an insecure mechanism unless it's combined with TLS/SSL. Consider setting up OpenSMTPD with SSL/TLS for encrypted connections.
- Make sure to use strong and unique passwords and consider using a mechanism to rotate them regularly.



You:

/etc/mail/credentials is empty. I keep my credentials in the file /etc/smtpd\_secrets



ChatGPT:

Alright, if you already have a file /etc/smtpd\_secrets where you store your credentials, then you can modify the OpenSMTPD configuration to point to that file instead.

#### 1. Update OpenSMTPD Configuration:

Modify /etc/smtpd.conf:

```
bash
table credentials file:/etc/smtpd_secrets
```

Make sure that the smtpd\_secrets file format follows the standard:

```
bash
billing@plan6.com fakepassword
```

#### 2. Ensure Proper Permissions:

Make sure that only the root user can read the /etc/smtpd\_secrets file to ensure security:

```
bash
sudo chmod 600 /etc/smtpd_secrets
sudo chown root:root /etc/smtpd_secrets
```

#### 3. Restart OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

#### 4. Testing from an External Machine:

After setting up SMTP authentication on your server, use swaks on an external machine:

```
bash
swaks --to twcookson@gmail.com --from billing@plan6.com --server 74.208.253.216 --auth-user billing@plan6.com
```

This should now work correctly with the provided credentials. If you run into any issues, you might want to check the logs for any errors related to authentication.



You:

Sep 03 19:41:48 plan6.com smtpd[5860]: /etc/smtpd.conf:21: syntax error Sep 03 19:41:48 plan6.com systemd[1]: opensmtpd.service: Control process exited, code=exited, status=1/FAILURE Sep 03 19:41:48 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:41:48 plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server. Sep 03 19:41:51 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:41:51 plan6.com smtpd[5864]: /etc/smtpd.conf:21: syntax error Sep 03 19:41:51 plan6.com systemd[1]: opensmtpd.service: Control process exited, code=exited, status=1/FAILURE Sep 03 19:41:51 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:41:51 plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server. lines 42-87/87 (END) With this config: table aliases file:/etc/aliases table credentials file:/etc/smtpd\_secrets table secrets file:/etc/smtpd\_secrets # To accept external mail, replace with: listen on all # listen on 74.208.253.216 listen on 0.0.0.0 port 25 listen on :: port 25 action "local" maildir alias <aliases> action "relay" relay action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets> accept from any for domain "plan6.com" relay accept for local deliver to mbox accept for any relay via smtp+auth://[127.0.0.1] auth <credentials> # Uncomment the following to accept external mail for domain "example.org" ## match from any for domain "example.org" action "local" match for local action "local" match from local for any action "relay" match auth from any for any action "authenticated" #net.ipv6.conf.all.disable\_ipv6 = 1 #net.ipv6.conf.default.disable\_ipv6 = 1 #net.ipv6.conf.lo.disable\_ipv6 = 1#



ChatGPT:

The configuration syntax error usually stems from using a directive or a format that OpenSMTPD doesn't recognize or expects differently.

Here's a more streamlined version of your configuration:

```
bash
table aliases file:/etc/aliases
table secrets file:/etc/smtpd_secrets

listen on 74.208.253.216
listen on 0.0.0.0 port 25
listen on :: port 25

action "local" maildir
action "relay" relay
action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets>

accept for domain "plan6.com" relay
accept for local deliver to mbox
accept from any for any relay via smtp+auth://[127.0.0.1] auth <secrets>
match for local action "local"
match from local for any action "relay"
match auth from any for any action "authenticated"
```

I removed the redundancy from the configuration. If you've already defined a table as secrets pointing to /etc/smtpd\_secrets, you can reference this table as <secrets> throughout your configuration.

Once you've updated the configuration, try restarting OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

If it still fails to start, the logs should provide more details about the cause of the failure.



Sep 03 19:45:39 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:45:39 plan6.com smtpd[5909]: /etc/smtpd.conf:12: syntax error  
Sep 03 19:45:39 plan6.com systemd[1]: opensmtpd.service: Control process exited, code=exited, status=1/FAILURE Sep 03 19:45:39 plan6.com  
systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:45:39 plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server. With table  
aliases file:/etc/aliases table secrets file:/etc/smtpd\_secrets listen on 74.208.253.216 listen on 0.0.0.0 port 25 listen on :: port 25 action "local" maildir  
action "relay" relay action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets> accept for domain "plan6.com" relay accept  
for local deliver to mbox accept from any for any relay via smtp+auth://[127.0.0.1] auth <secrets> match for local action "local" match from local for any  
action "relay" match auth from any for any action "authenticated" Perhaps replacing local host with 74.208.253.216 might work since that's solved other  
issues



The syntax error at line 12 suggests that the configuration format for one of the directives is incorrect.

The action "authenticated" line seems to be the cause. OpenSMTPD usually expects a relay via an external SMTP server. However, the provided configuration is trying to relay authenticated email back to the same server (localhost).

Also, if billing@plan6.com isn't an email account you have set up with an external provider (like Gmail or another SMTP server), then relaying to that email address on localhost doesn't make much sense.

Here's a slightly modified configuration:

```
plaintext
table aliases file:/etc/aliases
table secrets file:/etc/smtpd_secrets

listen on 74.208.253.216
listen on 0.0.0.0 port 25
listen on :: port 25

action "local" maildir
action "relay" relay
```

```
# For authenticated relay, you need an external SMTP provider details like smtp.gmail.com
# action "authenticated" relay host smtp+auth://yourSMTPUser@yourSMTPServer auth <secrets>

accept for domain "plan6.com" relay
accept for local deliver to mbox
accept from any for any relay
match for local action "local"
match from local for any action "relay"
```

I've commented out the authenticated action, but if you want to relay email through an external SMTP provider, you'll need to fill in the correct SMTP server and credentials.

Make these changes, save the configuration file, and try restarting OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

If you wish to use authentication, please provide details about your external SMTP provider (without revealing sensitive info).



You:

```
Sep 03 19:47:49 plan6.com smtpd[5920]: /etc/smtpd.conf:14: syntax error
Sep 03 19:47:49 plan6.com systemd[1]: opensmtpd.service: Control process
exited, code=exited, status=1/FAILURE
Sep 03 19:47:49 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'.
Sep 03 19:47:49
plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server. with table aliases file:/etc/aliases table secrets file:/etc/smtpd_secrets listen on
74.208.253.216 listen on 0.0.0.0 port 25 listen on :: port 25 action "local" maildir action "relay" relay # For authenticated relay, you need an external SMTP
provider details like smtp.gmail.com # action "authenticated" relay host smtp+auth://yourSMTPUser@yourSMTPServer auth <secrets> accept for domain
"plan6.com" relay accept for local deliver to mbox accept from any for any relay match for local action "local" match from local for any action "relay"
```



ChatGPT:

The syntax error suggests there's still something amiss in your configuration file.

Let's simplify this further:

1. First, we'll make sure you're able to receive emails on `plan6.com` without any authentication. Once that's confirmed, we can layer in complexities like relaying authenticated emails through an external SMTP provider.

Here's a basic configuration:

```
plaintext
table aliases file:/etc/aliases

listen on 74.208.253.216
listen on 0.0.0.0 port 25
listen on :: port 25

action "local" mbox
action "relay" relay

# Rules
accept for domain "plan6.com" deliver to mbox
accept from local for any relay
```

A few things to note:

- We've stripped out any mentions of `smtpd_secrets` for now.
- We're using `deliver to mbox` for local emails. This assumes that you've got local mailboxes set up.

Try with the above configuration:

1. Save it.
2. Restart OpenSMTPD with `sudo systemctl restart opensmtpd`.
3. Verify that the service is running without errors using `sudo systemctl status opensmtpd`.

Then try sending an email again from a remote system using `swaks`.

Once we're sure this basic configuration is functioning, we can reintroduce complexities, one step at a time.





You:

Sep 03 19:49:41 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:49:41 plan6.com smtpd[5933]: /etc/smtpd.conf:11: syntax error  
Sep 03 19:49:41 plan6.com smtpd[5933]: warn: no rules, nothing to do Sep 03 19:49:41 plan6.com systemd[1]: opensmtpd.service: Control process  
exited, code=exited, status=1/FAILURE Sep 03 19:49:41 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:49:41  
plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server. on tom@plan6:/etc\$ cat /etc/smtpd.conf table aliases file:/etc/aliases listen on  
74.208.253.216 listen on 0.0.0.0 port 25 listen on :: port 25 action "local" mbox action "relay" relay # Rules accept for domain "plan6.com" deliver to mbox  
accept from local for any relay



ChatGPT:

The syntax error you're seeing is likely from a directive that's not recognized or improperly formatted.

Let's address it:

1. The action "local" mbox may be incorrect. The correct format should be action "local" mbox alias <aliases>.
2. OpenSMTPD is very specific about syntax, including spaces. Ensure you're not accidentally using any tabs.

Here's a revised configuration:

```
plaintext
table aliases file:/etc/aliases

listen on 74.208.253.216 port 25
listen on 0.0.0.0 port 25
listen on :: port 25

action "local" mbox alias <aliases>
action "relay" relay

# Rules
accept for domain "plan6.com" deliver to mbox
accept from local for any relay
```

Save this configuration:

1. Restart OpenSMTPD with `sudo systemctl restart opensmtpd`.
2. Verify the status with `sudo systemctl status opensmtpd`.

If there's an error, please provide the exact line number mentioned in the syntax error message. This can help pinpoint which directive or line is causing the issue.



You:

tom@plan6:/etc\$ cat /etc/smtpd.conf table aliases file:/etc/aliases listen on 74.208.253.216 port 25 listen on 0.0.0.0 port 25 listen on :: port 25 action  
"local" mbox alias <aliases> action "relay" relay # Rules accept for domain "plan6.com" deliver to mbox accept from local for any relay tom@plan6:/etc\$  
Sep 03 19:51:54 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:51:54 plan6.com smtpd[5945]: /etc/smtpd.conf:11: syntax error  
Sep 03 19:51:54 plan6.com smtpd[5945]: warn: no rules, nothing to do Sep 03 19:51:54 plan6.com systemd[1]: opensmtpd.service: Control process  
exited, code=exited, status=1/FAILURE Sep 03 19:51:54 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:51:54  
plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server. lines 85-130/130 (END)



ChatGPT:

The issue seems to be with the line:

```
bash
action "local" mbox alias <aliases>
```

In OpenSMTPD, the correct way to specify local delivery to mailboxes is with `maildir` or `mbox`, but `mbox` doesn't take an alias argument.

Instead, try the following:

```
bash
```

```
action "local" maildir alias <aliases>
```

Or, if you're not using maildir:

```
arduino
action "local" mbox
```

Let's correct the configuration:

```
plaintext
table aliases file:/etc/aliases

listen on 74.208.253.216 port 25
listen on 0.0.0.0 port 25
listen on :: port 25

action "local" maildir
action "relay" relay

# Rules
accept for domain "plan6.com" deliver to maildir
accept from local for any relay
```

After updating the configuration:

1. Restart OpenSMTPD with `sudo systemctl restart opensmtpd`.
2. Check the status with `sudo systemctl status opensmtpd`.

If there's an error, provide the line number from the error message.



```
tom@plan6:/etc$ cat /etc/smtpd.conf table aliases file:/etc/aliases listen on 74.208.253.216 port 25 listen on 0.0.0.0 port 25 listen on :: port 25 action
"local" maildir action "relay" relay # Rules accept for domain "plan6.com" deliver to maildir accept from local for any relay tom@plan6:/etc$ Sep 03
19:54:46 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 19:54:46 plan6.com smtpd[5967]: /etc/smtpd.conf:11: syntax error Sep 03
19:54:46 plan6.com smtpd[5967]: warn: no rules, nothing to do Sep 03 19:54:46 plan6.com systemd[1]: opensmtpd.service: Control process exited,
code=exited, status=1/FAILURE Sep 03 19:54:46 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 19:54:46 plan6.com
systemd[1]: Failed to start OpenSMTPD SMTP server.
```



Alright, the persistence of a syntax error at line 11 is puzzling. Let's simplify this even further to get a basic working configuration.

1. Let's get rid of the `table aliases` since it seems you're not referencing it in your rules.
2. Let's fix the delivery directive for your domain.

Here's the simplified configuration:

```
plaintext
listen on 74.208.253.216 port 25
listen on 0.0.0.0 port 25
listen on :: port 25

action "local" maildir
action "relay" relay

# Rules
accept for domain "plan6.com" action "local"
accept from local for any action "relay"
```

Please update your `/etc/smtpd.conf` with this configuration and then attempt to restart the service again.



smtpplib.SMTPNotSupportedError: SMTP AUTH extension not supported by server. How do I enable this on ompensmtpd



If you're getting an SMTPNotSupportedError error about SMTP AUTH, it means that authentication hasn't been set up on your OpenSMTPD server. Let's get that sorted out:

1. **Set up your Secrets Table:** Make sure you have a table defined for your secrets in `/etc/smtpd.conf`. Since you mentioned your secrets are stored in `/etc/smtpd_secrets`, ensure that file is properly formatted, like:

```
scss
billing@plan6.com:fakepassword
```

Then in your `/etc/smtpd.conf`:

```
plaintext
table secrets file:/etc/smtpd_secrets
```

2. **Enable Authentication for your Listen Directives:**

Modify your listen directives to include the auth keyword:

```
plaintext
listen on 74.208.253.216 port 25 auth <secrets>
listen on 0.0.0.0 port 25 auth <secrets>
```

3. **Set up an Authenticated Relay Action:**

```
plaintext
action "authenticated_relay" relay auth <secrets>
```

4. **Modify Rules to Use Authenticated Relay:**

```
plaintext
match auth from any for any action "authenticated_relay"
```

5. **Finally, your configuration should look something like this:**

```
plaintext
table secrets file:/etc/smtpd_secrets

listen on 74.208.253.216 port 25 auth <secrets>
listen on 0.0.0.0 port 25 auth <secrets>
listen on :: port 25

action "local" maildir
action "relay" relay
action "authenticated_relay" relay auth <secrets>

# Rules
accept for domain "plan6.com" action "local"
accept from local for any action "relay"
match auth from any for any action "authenticated_relay"
```

6. **Restart OpenSMTPD:**

```
bash
sudo systemctl restart opensmtpd
```

Now, your SMTP server should support authentication. Test your server with a client that supports SMTP authentication to ensure it works.



invalid listen option: auth requires tls/smtps



Alright, the error you're seeing means that OpenSMTPD requires you to have TLS set up to use the authentication option. Here's what you need to do:

#### 1. Obtain an SSL/TLS Certificate:

If you don't already have a certificate for your domain, you can obtain a free certificate from [Let's Encrypt](#). If you have a certificate, you'll have a .crt (or .pem) and a .key file.

#### 2. Configure OpenSMTPD to use TLS:

Assuming you have your certificate and private key ready, specify their paths in your configuration. Add these lines to your `smtpd.conf`:

```
plaintext
pki plan6.com cert "/path/to/your_certificate.crt"
pki plan6.com key "/path/to/your_private_key.key"
```

Replace `/path/to/your_certificate.crt` and `/path/to/your_private_key.key` with the actual paths to your certificate and key files.

#### 3. Update your Listen Directives:

Change your listen directives to look like the following:

```
plaintext
listen on 74.208.253.216 port 25 tls pki plan6.com auth <secrets>
listen on 0.0.0.0 port 25 tls pki plan6.com auth <secrets>
```

The `tls` keyword tells OpenSMTPD to use TLS, and the `pki plan6.com` refers to the certificate and key you specified.

#### 4. Complete Configuration:

The rest of your configuration will remain the same:

```
plaintext
action "local" maildir
action "relay" relay
action "authenticated_relay" relay auth <secrets>

# Rules
accept for domain "plan6.com" action "local"
accept from local for any action "relay"
match auth from any for any action "authenticated_relay"
```

#### 5. Restart OpenSMTPD:

```
bash
sudo systemctl restart opensmtpd
```

After these steps, your OpenSMTPD server should support both TLS and authentication. Remember, using TLS not only enables authentication but also encrypts the emails in transit, which is essential for security.



You:

I have let's encrypt working on html for plan6.com's domain, and the folder where the files are on this webpage. How do I find the .crt and .key, assuming a https cert works



If you're using Let's Encrypt, the standard paths for the certificate and key files are:

- **Certificate file:** `/etc/letsencrypt/live/plan6.com/fullchain.pem`
- **Private key file:** `/etc/letsencrypt/live/plan6.com/privkey.pem`

However, it's essential to ensure the paths are correct. You can check the existence of these files with:

```
bash
ls /etc/letsencrypt/live/plan6.com/
```

If you see `fullchain.pem` and `privkey.pem` in the output, those are the files you need.

Once you have identified the paths to your certificate and key files, update your OpenSMTPD configuration (`smtpd.conf`) accordingly:

```
plaintext
pki plan6.com cert "/etc/letsencrypt/live/plan6.com/fullchain.pem"
pki plan6.com key "/etc/letsencrypt/live/plan6.com/privkey.pem"
```

And then use the `tls` and `pki` options in your `listen` directives as previously mentioned.

Also, note that Let's Encrypt certificates are only valid for 90 days, so they need to be renewed regularly. If you're using `certbot` (a tool from Let's Encrypt) for certificate issuance, it might have set up a cron job or systemd timer for automatic renewal. If it hasn't, you'll want to do so to ensure your certificate doesn't expire.

-----



Before I proceed this looks right?: tom@plan6:/etc/letsencrypt\$ sudo ls live/plan6.com cert.pem chain.pem fullchain.pem privkey.pem README  
tom@plan6:/etc/letsencrypt\$

-----



Yes, that looks correct. The important files for you are:

- **fullchain.pem**: This includes your domain's certificate along with any intermediate certificates. It's the certificate file you'll typically use in server configurations.
- **privkey.pem**: This is the private key for your certificate.

So, you'll use these paths in your OpenSMTPD configuration:

```
plaintext
pki plan6.com cert "/etc/letsencrypt/live/plan6.com/fullchain.pem"
pki plan6.com key "/etc/letsencrypt/live/plan6.com/privkey.pem"
```

Ensure you reference these files in your `smtpd.conf` when configuring TLS for your mail server. Remember to restart or reload OpenSMTPD after making any configuration changes for them to take effect.

-----



Changing as little as possible (some things that look like mistakes are intentional) add those and what else is absolutely necessary to this file: table aliases file:/etc/aliases table secrets file:/etc/smtpd\_secrets # Listen on all available network interfaces, you can change it to "listen on localhost" if you only want to allow local connections listen on 74.208.253.216 port 25 auth <secrets> action "local" maildir alias <aliases> action "relay" relay action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets> match for local action "local" match from local for any action "relay" match auth from any for any action "authenticated" # Rules accept for domain "plan6.com" deliver to maildir accept from local for any relay

-----



To enable TLS using Let's Encrypt on your OpenSMTPD configuration, we'll need to:

1. Add the `pki` directives to specify where the certificates are.
2. Adjust the `listen` directive to enable TLS/SMTPS and specify which certificate (by its domain name) to use.

Here's the modified `smtpd.conf`:

```
plaintext
# Define the location of your certificates
pki plan6.com cert "/etc/letsencrypt/live/plan6.com/fullchain.pem"
pki plan6.com key "/etc/letsencrypt/live/plan6.com/privkey.pem"
```

```

table aliases file:/etc/aliases
table secrets file:/etc/smtpd_secrets

# Listen for regular SMTP traffic on port 25 with auth
listen on 74.208.253.216 port 25 auth <secrets> hostname plan6.com

# Additionally, listen for SMTPS traffic on port 465 (this is the usual port for SMTPS) with the certificate
listen on 74.208.253.216 port 465 auth <secrets> tls pki plan6.com hostname plan6.com

action "local" maildir alias <aliases>
action "relay" relay
action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets>

# Rules
accept for domain "plan6.com" deliver to maildir
accept from local for any relay
match for local action "local"
match from local for any action "relay"
match auth from any for any action "authenticated"

```

Remember:

1. Always backup your configuration files before making changes.
2. After making these changes, be sure to restart OpenSMTPD to apply the new settings.
3. Ensure your firewall settings allow traffic on port 465 if you're enabling SMTPS.

Lastly, note that I added `hostname plan6.com` in the `listen` directive. This helps ensure that the server identifies itself correctly during SMTP transactions. It's particularly important when using TLS.



You:

this: # Define the location of your certificates pki plan6.com cert "/etc/letsencrypt/live/plan6.com/fullchain.pem" pki plan6.com key "/etc/letsencrypt/live/plan6.com/privkey.pem" table aliases file:/etc/aliases table secrets file:/etc/smtpd\_secrets # Listen for regular SMTP traffic on port 25 with auth listen on 74.208.253.216 port 25 auth <secrets> hostname plan6.com # Additionally, listen for SMTPS traffic on port 465 (this is the usual port for SMTPS) with the certificate listen on 74.208.253.216 port 465 auth <secrets> tls pki plan6.com hostname plan6.com action "local" maildir alias <aliases> action "relay" relay action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets> # Rules accept for domain "plan6.com" deliver to maildir accept from local for any relay match for local action "local" match from local for any action "relay" match auth from any for any action "authenticated" Causes Sep 03 20:24:37 plan6.com systemd[1]: Starting OpenSMTPD SMTP server... Sep 03 20:24:37 plan6.com smtpd[6146]: smtpd: invalid listen option: auth requires tls/smtps Sep 03 20:24:37 plan6.com systemd[1]: opensmtpd.service: Control process exited, code=exited, status=1/FAILURE Sep 03 20:24:37 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 20:24:37 plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server.



ChatGPT:

I apologize for the oversight. The `auth` option in the `listen` directive requires TLS, which makes sense from a security perspective. You should not have authentication without encryption.

Here's how to fix it:

1. Remove the `auth` option from the `listen on 74.208.253.216 port 25` line because it's intended for unencrypted SMTP.
2. Ensure `auth` is only specified in the `listen` directive that has `tls` for encrypted SMTP (SMTPS on port 465).

Here's the corrected configuration:

```

plaintext
# Define the location of your certificates
pki plan6.com cert "/etc/letsencrypt/live/plan6.com/fullchain.pem"
pki plan6.com key "/etc/letsencrypt/live/plan6.com/privkey.pem"

table aliases file:/etc/aliases
table secrets file:/etc/smtpd_secrets

# Listen for regular SMTP traffic on port 25 without auth
listen on 74.208.253.216 port 25 hostname plan6.com

# Listen for SMTPS traffic on port 465 with the certificate and auth
listen on 74.208.253.216 port 465 auth <secrets> tls pki plan6.com hostname plan6.com

action "local" maildir alias <aliases>
action "relay" relay
action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets>

```

```
# Rules
accept for domain "plan6.com" deliver to maildir
accept from local for any relay
match for local action "local"
match from local for any action "relay"
match auth from any for any action "authenticated"
```

Now, after making this change, you can restart OpenSMTPD and it should work correctly.



This: # Define the location of your certificates pki plan6.com cert "/etc/letsencrypt/live/plan6.com/fullchain.pem" pki plan6.com key "/etc/letsencrypt/live/plan6.com/privkey.pem" table aliases file:/etc/aliases table secrets file:/etc/smtpd\_secrets # Listen for regular SMTP traffic on port 25 without auth listen on 74.208.253.216 port 25 hostname plan6.com # Listen for SMTPS traffic on port 465 with the certificate and auth listen on 74.208.253.216 port 465 auth <secrets> tls pki plan6.com hostname plan6.com action "local" maildir alias <aliases> action "relay" relay action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets> # Rules accept for domain "plan6.com" deliver to maildir accept from local for any relay match for local action "local" match from local for any action "relay" match auth from any for any action "authenticated" causes Sep 03 20:26:36 plan6.com smtpd[6164]: /etc/smtpd.conf:19: syntax error Sep 03 20:26:36 plan6.com systemd[1]: opensmtpd.service: Control process exited, code=exited, status=1/FAILURE Sep 03 20:26:36 plan6.com systemd[1]: opensmtpd.service: Failed with result 'exit-code'. Sep 03 20:26:36 plan6.com systemd[1]: Failed to start OpenSMTPD SMTP server



I apologize for the inconvenience.

The error suggests a syntax issue in the config, specifically on line 19. To narrow down the problem, let's isolate and check that line.

Given your provided configuration, line 19 is:

```
plaintext
action "authenticated" relay host smtp+auth://billing@plan6.com@localhost auth <secrets>
```

Upon examining it, the issue appears to be with the smtp+auth URL. Having a double "@" in the URL (billing@plan6.com@localhost) is unconventional and may be causing the problem.

You'll typically specify the SMTP user and then the SMTP server, separated by "@". If billing@plan6.com is your username for authentication, and localhost is your SMTP server, then you've essentially combined them in a way that's causing confusion.

To fix:

1. Clarify your SMTP server and the username you're using for SMTP authentication.
2. Adjust the smtp+auth URL accordingly.

For example, if you're trying to relay through Gmail, you'd use something like:

```
plaintext
action "authenticated" relay host smtp+auth://username@gmail.com:password@smtp.gmail.com auth <secrets>
```

Replace username@gmail.com with your actual email and password with your actual password. Adjust the SMTP server (after the "@") as per your actual SMTP server.

Given you mentioned localhost earlier, if you're trying to relay through a local SMTP server that requires the username billing@plan6.com for authentication, then you have to reformat it properly.

However, if billing@plan6.com@localhost was an attempt to specify both the username and the SMTP server in one go, you'll need to adjust that.

Once you make the corrections, restart OpenSMTPD and check if the issue is resolved.



What does this mean to you: "failed-command command="AUTH NTLM TIRMTVNTUAABAAAAB4IIAAAAAAAAAAAAAAAAAAAAAAAAAAAA=" result="503 5.5.1 Invalid command: Command not supported"

-----

The error message:

```
bash
"failed-command command="AUTH NTLM TlRMTVNTUAABAAAAB4IIAAAAAAAAAAAAAAAAAAAAA=" result="503 5.5.1 Invalid command"
```

indicates an attempt to authenticate to an SMTP server using the NTLM (NT LAN Manager) authentication mechanism. However, the server responded with a "503 5.5.1 Invalid command: Command not supported" error. This suggests the server does not support or is not configured to handle NTLM authentication.

Here's a breakdown:

1. **AUTH NTLM:** This is an SMTP command where the client tells the server that it wants to authenticate using the NTLM mechanism.
2. **TlRMTVNTUAABAAAAB4IIAAAAAAAAAAAAAAAAAAAAA=:** This looks like a base64 encoded initial message in the NTLM authentication handshake. It's essentially a part of the NTLM challenge-response mechanism.
3. **503 5.5.1 Invalid command:** This is the SMTP server's response, indicating that the command given (in this case, NTLM authentication) is not recognized or supported.

What it means for you:

- The SMTP server you're trying to authenticate to doesn't support NTLM authentication, or it isn't configured to handle it.
- You should check if the server supports other authentication mechanisms (like PLAIN, LOGIN, etc.) and configure your client accordingly.
- If you control the SMTP server, ensure that NTLM authentication is enabled and properly configured, especially if you need to use it.
- If you're using a third-party service, check their documentation or contact their support to find out the supported authentication methods.



You:

I'm getting there error from this: 

```
import smtplib from email.message import EmailMessage def send_email(): # Define the email parameters smtp_server = "plan6.com" smtp_port = 25 sender_email = "billing@plan6.com" sender_password = "redacted" recipient_email = "twcookson@gmail.com" subject = "Test Email 2" body = "This is a test email sent from Python." # Create the email msg = EmailMessage() msg.set_content(body) msg["Subject"] = subject msg["From"] = sender_email msg["To"] = recipient_email # Send the email with smtplib.SMTP(smtp_server, smtp_port) as server: # Uncomment the next line if you're using SSL # server.starttls() server.login(sender_email, sender_password) server.send_message(msg) print("Email sent successfully!") send_email()
```

 How do I simplify the method



ChatGPT:

The error you mentioned previously, "failed-command command="AUTH NTLM ... result="503 5.5.1 Invalid command: Command not supported", indicates that the server doesn't support NTLM authentication. It seems you're trying to send an email via an SMTP server, and Python's smtplib might be defaulting to NTLM or another authentication method that the server doesn't recognize.

To simplify and potentially fix the code, you can make sure to use a standard authentication method, typically PLAIN or LOGIN.

Here's a revised version of your function:

```
python
import smtplib
from email.message import EmailMessage

def send_email():
    # Define the email parameters
    smtp_server = "plan6.com"
    smtp_port = 25
    sender_email = "billing@plan6.com"
    sender_password = "redacted"
    recipient_email = "twcookson@gmail.com"
    subject = "Test Email 2"
    body = "This is a test email sent from Python."

    # Create the email
    msg = EmailMessage()
    msg.set_content(body)
    msg["Subject"] = subject
    msg["From"] = sender_email
    msg["To"] = recipient_email

    # Send the email
```



```

with smtplib.SMTP(smtp_server, smtp_port) as server:
    # If you want to use SSL
    # server = smtplib.SMTP_SSL(smtp_server, 465)
    # server.login(sender_email, sender_password)

    # StartTLS can be an option for encryption if the server supports it
    server.starttls()
    server.login(sender_email, sender_password)
    server.send_message(msg)

print("Email sent successfully!")

send_email()

```

You can uncomment the section for `smtplib.SMTP_SSL` if you want to use SSL (typically on port 465). The `starttls()` method is used to upgrade a plain SMTP connection to an encrypted TLS (or SSL) connection. If your server supports it, it's a good idea to use it.



You:

lets remove opensmtpd from the server and try something else



ChatGPT:

Sure, if you're finding OpenSMTPD problematic for your needs, there are other SMTP server options available. Here are a few popular ones:

1. **Postfix:** A very common MTA (Mail Transfer Agent) known for its high reliability and configurability.
2. **Exim:** Another widely used MTA with a focus on flexibility.
3. **Sendmail:** One of the oldest and well-known MTAs. However, it's often considered complex to set up and manage.
4. **Dovecot:** Mainly an IMAP and POP3 server, but can work in conjunction with Postfix for SMTP.

I'd recommend **Postfix** if you're looking for a balance between features, flexibility, and ease of use.

To install and set up Postfix:

#### 1. Uninstall OpenSMTPD:

```

bash
sudo apt-get purge opensmtpd

```

#### 2. Install Postfix:

```

bash
sudo apt-get install postfix

```

During the installation, you'll be presented with a series of prompts to configure your server. For a basic setup, the following should work:

- General type of mail configuration: Internet Site
- System mail name: (Your domain, e.g., plan6.com)

#### 3. Secure Postfix:

By default, Postfix will operate as an open relay, which means anyone can send email through your server. You need to lock it down.

Edit the main Postfix configuration file:

```

bash
sudo nano /etc/postfix/main.cf

```

And make sure you have the following settings:

```

bash
mydestination = $myhostname, localhost.$mydomain, $mydomain
inet_interfaces = all
inet_protocols = ipv4
mynetworks = 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128

```

Restart Postfix to apply the changes:

```

bash

```

```
sudo systemctl restart postfix
```

4. **Test Your Configuration:** Try sending a test email using the command line:

```
bash
echo "Test email from Postfix" | mail -s "Test Postfix" you@example.com
```

5. If you want more features, like authentication, SSL/TLS support, or spam filtering, you'll need to do some additional configuration. This is a basic setup to get you started.

Would you like more detailed steps or guidance on setting up some of these additional features?



Let me send a email to it from an external python client



Sure! If you've set up Postfix and it's currently listening on port 25, you should be able to send an email to it from an external client, provided there are no firewalls blocking the connection.

Here's a simple Python script using the `smtpplib` and `email.message` modules to send an email:

```
python
import smtplib
from email.message import EmailMessage

def send_email():
    # Define the email parameters
    smtp_server = "plan6.com"
    smtp_port = 25 # Default port for SMTP
    sender_email = "you@externaldomain.com"
    recipient_email = "billing@plan6.com"
    subject = "Test Email from External Client"
    body = "This is a test email sent from an external Python client."

    # Create the email
    msg = EmailMessage()
    msg.set_content(body)
    msg["Subject"] = subject
    msg["From"] = sender_email
    msg["To"] = recipient_email

    # Send the email
    with smtplib.SMTP(smtp_server, smtp_port) as server:
        server.send_message(msg)

    print("Email sent successfully!")

send_email()
```

Replace "you@externaldomain.com" with your own sender email address.

Please note:

1. If you've set up Postfix as mentioned in the previous instructions, you won't need authentication to send the email. The provided script is for a non-authenticated SMTP connection.
2. If you have firewalls or other security mechanisms in place on your plan6.com server, you might need to allow connections on port 25 from the client machine.
3. Make sure DNS records for plan6.com are pointing correctly to your server.

Once you run the script, check the mailbox on the plan6.com server to see if you received the email. If it's not there, inspect the mail logs on the server (typically `/var/log/mail.log` on Ubuntu systems) to check for any errors or issues.



smtplib.SMTPRecipientsRefused: {'twcookson@gmail.com': (454, b'4.7.1 <twcookson@gmail.com>: Relay access denied')}



The error `smtplib.SMTPRecipientsRefused` with the message `Relay access denied` indicates that the SMTP server (in this case, your Postfix server) is refusing to relay the email to an external domain (gmail.com).

By default, Postfix does not allow arbitrary clients to relay mail through it. This is a security feature to prevent unauthorized use of the SMTP server (e.g., sending spam). Relaying means accepting mail from one domain and forwarding it to another.

To fix this:

1. **Allow Relaying for Specific IP Addresses:** If you trust the client IP address and want it to be able to relay through Postfix, you can add it to the `mynetworks` parameter in Postfix's main configuration file (`/etc/postfix/main.cf`).

```
css
mynetworks = 127.0.0.0/8, [::1]/128, <your_client_ip_address>/32
```

Replace `<your_client_ip_address>` with the IP address of the client machine from which you're trying to send email.

2. **Reload Postfix:** After making the change, reload the Postfix service:

```
bash
sudo systemctl reload postfix
```

3. **Authentication:** Alternatively, if you want to allow relay access based on authentication (so that only authenticated users can relay mail), you'll need to set up SMTP authentication. This is more complex and involves configuring SASL authentication for Postfix.

For the sake of simplicity and if you trust the client IP, you can use the first solution. However, if you're setting up a production server or want to allow multiple users to send emails, consider setting up SMTP authentication.

Always be cautious when allowing relay access, as misconfigurations can turn your SMTP server into an open relay, which can be abused by spammers.



You:

Authentication: Alternatively, if you want to allow relay access based on authentication (so that only authenticated users can relay mail), you'll need to set up SMTP authentication. This is more complex and involves configuring SASL authentication for Postfix.



Alright, let's set up SMTP authentication for Postfix using SASL (Cyrus SASL in this case).

## Step 1: Install Required Packages

Install the necessary packages:

```
bash
sudo apt update
sudo apt install postfix sasl2-bin libsasl2-modules
```

## Step 2: Configuring SASL

Edit the SASL configuration file for Postfix:

```
bash
sudo nano /etc/default/saslauthd
```

Change the START option to yes:

```
bash
START=yes
```

Set the mechanism to `sasl_db`:

```
makefile
MECHANISMS="saslauthd"
```

Now, start the saslauthd service:

```
bash
sudo systemctl start saslauthd
sudo systemctl enable saslauthd
```

### Step 3: Add SASL Users

Use saslpasswd2 to add users. These users are stored in /etc/saslauthd2.

```
bash
sudo saslpasswd2 -c -u your_domain.com username
```

Replace your\_domain.com with your domain and username with your desired username. You'll be prompted to set a password for this user.

### Step 4: Configure Postfix for SASL

Edit the Postfix main configuration:

```
bash
sudo nano /etc/postfix/main.cf
```

Add or modify the following lines:

```
makefile
smtpd_sasl_auth_enable = yes
smtpd_sasl_type = cyrus
smtpd_sasl_path = smtpd
smtpd_sasl_security_options = noanonymous
smtpd_sasl_local_domain = $myhostname
smtpd_recipient_restrictions =
    permit_mynetworks,
    permit_sasl_authenticated,
    reject_unauth_destination
```

Now, tell Postfix where to find the SASL libraries:

```
bash
sudo postconf -e 'smtpd_sasl_path = /var/spool/postfix/var/run/saslauthd/'
```

### Step 5: Adjust Permissions and Groups

Add postfix user to the sasl group:

```
bash
sudo adduser postfix sasl
```

### Step 6: Restart Services

Now, restart Postfix and saslauthd to apply the changes:

```
bash
sudo systemctl restart postfix saslauthd
```

With these changes, Postfix should now support SMTP authentication. When you connect to the SMTP server to send mail, you should authenticate with the username and password you set using saslpasswd2.

Remember to secure your server properly. This setup does allow for basic email sending with authentication, but there are many additional configurations (like TLS for encrypted connections) you might want to consider for a production environment.

-----



You:

Give me something like this to test it: import smtplib from email.message import EmailMessage def send\_email(): # Define the email parameters smtp\_server = "plan6.com" smtp\_port = 25 # Default port for SMTP sender\_email = "you@externaldomain.com" recipient\_email = "billing@plan6.com"

```
subject = "Test Email from External Client" body = "This is a test email sent from an external Python client." # Create the email msg = EmailMessage()
msg.set_content(body) msg["Subject"] = subject msg["From"] = sender_email msg["To"] = recipient_email # Send the email with
smtpplib.SMTP(smtp_server, smtp_port) as server: server.send_message(msg) print("Email sent successfully!") send_email()
```



Sure, based on the given information and the SASL setup you've done on Postfix, here's a simple email script to test the authenticated relay:

```
python
import smtplib
from email.message import EmailMessage

def send_email():
    # Define the email parameters
    smtp_server = "plan6.com"
    smtp_port = 587 # Using 587 as it's the typical port for SMTP with STARTTLS (change if needed)
    sender_email = "username@plan6.com" # Replace 'username' with the SASL user you created
    sender_password = "your_password" # Replace with the password you set for the SASL user
    recipient_email = "billing@plan6.com"
    subject = "Test Email from External Client"
    body = "This is a test email sent from an external Python client."

    # Create the email
    msg = EmailMessage()
    msg.set_content(body)
    msg["Subject"] = subject
    msg["From"] = sender_email
    msg["To"] = recipient_email

    # Send the email
    with smtplib.SMTP(smtp_server, smtp_port) as server:
        server.starttls() # Upgrade the connection to SSL/TLS
        server.login(sender_email, sender_password) # Authenticate with the server
        server.send_message(msg)

    print("Email sent successfully!")

send_email()
```

Some key points:

1. Make sure the username@plan6.com is the SASL user you created.
2. Use the password you set for the SASL user.
3. I've used port 587 for STARTTLS. If you've set up your server differently, adjust as needed.
4. The server.starttls() call is to use STARTTLS, which upgrades a plain text connection to an encrypted one. If you're using pure SSL/TLS on port 465, you'd use smtpplib.SMTP\_SSL instead of smtpplib.SMTP, and you'd skip the starttls() call. Adjust according to your server setup.

Please replace placeholders like "username" and "your\_password" with the appropriate values. Run the script and see if the email is successfully relayed through your Postfix server.



Can I check it's actually listening on 587 before doing this?



Certainly! You can use the netstat command or the ss command (a replacement for netstat on many modern systems) to check if a process is listening on a specific port. Here's how to do it:

1. Using netstat:

```
bash
netstat -tuln | grep 587
```

2. Using ss:

```
bash
ss -tln | grep 587
```

If Postfix is indeed listening on port 587, you should see an output like:

```
ruby
tcp 0 0 0.0.0.0:587 0.0.0.0:* LISTEN
```

or similarly for IPv6:

```
ruby
tcp6 0 0 :::587 :::* LISTEN
```

If you see such an output, it means that Postfix (or some other service) is indeed listening on port 587. If there's no output, then nothing is currently listening on that port.

-----



Hmmm only seems to be listening on 25 right now

-----



If Postfix is only listening on port 25 and not on 587 (which is the standard port for SMTP with StartTLS), you'll need to adjust the Postfix configuration to listen on this port.

#### 1. Update `master.cf`:

First, open the `master.cf` file in Postfix's configuration directory (usually `/etc/postfix/master.cf`). Look for a section that starts with `submission`. If it's commented out (lines beginning with `#`), uncomment that section:

```
bash
submission inet n      -       y       -       -       smtpd
  -o syslog_name=postfix/submission
  -o smtpd_tls_security_level=encrypt
  -o smtpd_sasl_auth_enable=yes
  -o smtpd_tls_auth_only=yes
  ...
```

This section is typically included in default Postfix installations to configure port 587 for submission of mail by email clients, which is precisely what you want.

#### 2. Reload or restart Postfix:

After making the changes, you need to reload or restart Postfix to apply them:

```
bash
sudo systemctl reload postfix
```

#### 3. Check again:

Now you can use the `ss` or `netstat` command again to check if Postfix is listening on port 587:

```
bash
ss -tln | grep 587
```

If everything is configured correctly, you should see Postfix listening on port 587.

#### 4. Adjust your Python script:

Once Postfix is listening on port 587, you might also want to adjust your Python script to send mail to port 587 instead of port 25, especially if you're using StartTLS.

Remember: If you plan to use authentication, you'll also need to ensure that Postfix is set up correctly with SASL authentication and that you have valid credentials to use when sending mail.

-----



You:

```
Still not listening on 587: smtp inet n - y - - smtpd -o syslog_name=postfix/submission -o smtpd_tls_security_level=encrypt -o
smtpd_sasl_auth_enable=yes -o smtpd_tls_auth_only=yes flags=DRXhu user=vmail argv=/usr/bin/maildrop -d ${recipient} pickup unix n - y 60 1 pickup
cleanup unix n - y 0 cleanup qmgr unix n - n 300 1 qmgr #qmgr unix n - n 300 1 oqmgr tlsmgr unix - - y 1000? 1 tlsmgr rewrite unix - - y - - trivial-rewrite
bounce unix - - y 0 bounce defer unix - - y 0 bounce trace unix - - y 0 bounce verify unix - - y 1 verify flush unix n - y 1000? 0 flush proxymap unix - -
n - - proxymap proxywrite unix - - n 1 proxymap smtp unix - - y - - smtp relay unix - - y - - smtp -o syslog_name=postfix/$service_name # -o
smtp_helo_timeout=5 -o smtp_connect_timeout=5 showq unix n - y - - showq error unix - - y - - error retry unix - - y - - error discard unix - - y - - discard
local unix - n n - - local virtual unix - n n - - virtual lmtmp unix - - y - - lmtmp anvil unix - - y 1 anvil scache unix - - y 1 scache postlog unix-dgram n - n 1
postlogd maildrop unix - n n - - pipe uucp unix - n n - - pipe flags=Fqhu user=uucp argv=uux -r -n -z -a$sender - $nexthop!rmail ($recipient) ifmail unix - n n
- - pipe flags=F user=ftn argv=/usr/lib/iftmail/iftmail -r $nexthop ($recipient) bsmtp unix - n n - - pipe flags=Fq. user=bsmtp argv=/usr/lib/bsmtp/bsmtp -
t$nexthop -f$sender $recipient scalemail-backend unix - n n - 2 pipe flags=R user=scalemail argv=/usr/lib/scalemail/bin/scalemail-store ${nexthop} ${u>
mailman unix - n n - - pipe flags=FRX user=list argv=/usr/lib/mailman/bin/postfix-to-mailman.py ${nexthop} ${>
```



ChatGPT:

The configuration you provided for `master.cf` does not seem to explicitly specify that Postfix should listen on port 587 (submission). In a typical Postfix configuration, the `submission` line specifies that Postfix should listen on port 587 for SMTP with StartTLS.

Here's how you can adjust your `master.cf` to listen on port 587 for submission:

1. Find the line in your `master.cf` that reads:

```
bash
smtp      inet  n       -       y       -       -       smtpd
```

2. Right below it, add the following lines:

```
bash
submission inet n       -       y       -       -       smtpd
-o syslog_name=postfix/submission
-o smtpd_tls_security_level=encrypt
-o smtpd_sasl_auth_enable=yes
-o smtpd_reject_unlisted_recipient=no
-o smtpd_client_restrictions=$mua_client_restrictions
-o smtpd_helo_restrictions=$mua_helo_restrictions
-o smtpd_sender_restrictions=$mua_sender_restrictions
-o smtpd_recipient_restrictions=
-o smtpd_relay_restrictions=permit_sasl_authenticated,reject
-o milter_macro_daemon_name=ORIGINATING
```

3. Save the `master.cf` file.

4. Reload or restart Postfix:

```
bash
sudo systemctl restart postfix
```

5. Verify if Postfix is now listening on port 587 using the following command:

```
bash
ss -tln | grep 587
```

After these steps, Postfix should be listening on port 587 for submission. Once that is done, you can try sending an email to your server using port 587.



You:

I'll right, 587 is up locally and telnet from my PC works. Okay, right me a python script that will send an email with this server, from a different computer



ChatGPT:

Certainly! Below is a Python script that uses the `smtpplib` library to send an email through your server from an external machine:

```
python
import smtpplib
from email.message import EmailMessage

def send_email():
    # Define the email parameters
    smtp_server = "plan6.com"
    smtp_port = 587 # Port for SMTP submission
    sender_email = "you@externaldomain.com" # Change this to the sender's email address
    sender_password = "yourpassword" # Change this to the sender's email password
    recipient_email = "billing@plan6.com"
    subject = "Test Email from External Client"
    body = "This is a test email sent from an external Python client."

    # Create the email
    msg = EmailMessage()
    msg.set_content(body)
    msg["Subject"] = subject
    msg["From"] = sender_email
    msg["To"] = recipient_email

    # Send the email
    with smtpplib.SMTP(smtp_server, smtp_port) as server:
        server.starttls() # Upgrade the connection to SSL/TLS
        server.login(sender_email, sender_password) # Login to the SMTP server
        server.send_message(msg)

    print("Email sent successfully!")

send_email()
```

A few things to note:

1. Replace "you@externaldomain.com" with the sender's actual email address.
2. Replace "yourpassword" with the sender's actual email password.
3. This script assumes that your Postfix server on plan6.com supports and requires STARTTLS for secure connections on port 587.
4. Ensure that the you@externaldomain.com address is allowed to send emails through your Postfix server. You may need to adjust Postfix's SASL settings to authenticate this sender.

Always handle passwords with care. It's recommended not to hard-code them into scripts. Consider using environment variables or other methods to securely manage credentials.



How do I see the error log for the email: `smtpplib.SMTPAuthenticationError: (535, b'5.7.8 Error: authentication failed: authentication failure')`

-----



The error you're seeing, `smtpplib.SMTPAuthenticationError: (535, b'5.7.8 Error: authentication failed: authentication failure')`, is raised when the SMTP server rejects the provided credentials (username/password).

To diagnose the cause of the authentication error, you'll want to check the mail server's logs. Since you mentioned you're using Postfix:

1. **Check the logs:** The primary log file for Postfix is typically `/var/log/mail.log` on most systems. You can view the most recent entries with:

```
bash
tail -f /var/log/mail.log
```

Look for any `NOQUEUE: reject` or authentication failure related messages around the time you attempt to send an email from your Python script.

2. **Adjust the log level:** If you're not seeing helpful information, you can increase the log verbosity of the Postfix server to get more detailed logs about authentication:
  - Edit the Postfix main configuration file, usually located at `/etc/postfix/main.cf`.
  - Add or modify the line: `debug_peer_level = 2`
  - Restart the Postfix service: `sudo systemctl restart postfix`



Then, try sending the email again from your Python script and re-check the logs. The increased log verbosity might give you more clues.

### 3. Cross-check credentials and configuration:

- Ensure that you're using the correct username and password in your Python script.
- Verify that the username you're using is allowed to relay mail.
- Check the SASL authentication configuration on your Postfix server.

### 4. Security considerations:

It's important to note that passwords should never be hard-coded directly in scripts. If you need to use the script frequently or on different machines, consider implementing a more secure way to handle the credentials, like using environment variables or encrypted configuration files.

After checking the logs and potentially making changes, be sure to reset the `debug_peer_level` setting to its default value (usually 0) to avoid filling up your log files with verbose debug information.



`/var/log/mail.log` this doesn't exist

-----



If `/var/log/mail.log` doesn't exist, Postfix might be logging to a different location or might be using a different system logging mechanism.

Here's what you can do to locate or enable logging:

#### 1. Check Other Common Log Locations:

- `/var/log/maillog` (no dot between "mail" and "log")
- `/var/log/syslog` (Postfix messages can sometimes be logged here on certain systems)

Use the `tail` command to view the most recent entries for each:

```
bash
tail -f /var/log/maillog
```

or

```
bash
tail -f /var/log/syslog
```

#### 2. Check Postfix's Configuration for Log Settings:

- Open `/etc/postfix/main.cf` and look for any settings related to logging or syslog.

#### 3. Syslog Configuration:

Postfix usually logs messages via the system logging daemon, `syslogd`. Depending on your system, the configuration for `syslog` might be in `/etc/syslog.conf`, `/etc/rsyslog.conf`, or another location if you're using a different logging system like `journald`.

- For `rsyslog`, check `/etc/rsyslog.conf` and see where mail-related messages are being directed.
- For `syslog-ng`, the configuration might be in `/etc/syslog-ng/syslog-ng.conf`.
- If you're using `journald` (common with `systemd`), you can check Postfix logs with:

```
bash
journalctl -u postfix
```

#### 4. Ensure Logging is Enabled:

If you still can't find any logs, it's possible that logging is not correctly configured for Postfix or the `syslog` daemon itself. In such cases, you might need to reconfigure or restart the logging daemon.

Lastly, if you have made any changes to logging configurations, ensure you restart the associated services (`postfix`, `rsyslog`, `syslog-ng`, etc.) for changes to take effect.



Hmmm nothing worked

-----

Sorry to hear that. The absence of logging can make debugging especially challenging.

Let's try a couple of other diagnostics:

1. **Postfix Status:** First, let's ensure Postfix is running and active.

```
bash
sudo systemctl status postfix
```

This command should give you some insights into whether Postfix is active, has encountered errors, or has been restarted multiple times.

2. **Manual Mail Testing:** You can use the `mail` command to send a test email. This can sometimes produce an immediate error message that provides a clue. Install `mailutils` if you don't have it already:

```
bash
sudo apt-get install mailutils
```

Then, send a test email:

```
bash
echo "Test email body" | mail -s "Test Subject" your_email@example.com
```

3. **Direct Postfix Logging:** We can increase the verbosity of Postfix for a short duration to see if we get more detailed logs. First, back up your `main.cf`:

```
bash
sudo cp /etc/postfix/main.cf /etc/postfix/main.cf.backup
```

Then, edit `/etc/postfix/main.cf` and add:

```
bash
debug_peer_level = 2
```

Restart Postfix:

```
bash
sudo systemctl restart postfix
```

Now, try sending an email again. See if this verbosity level produces any logs. After the test, you can revert the changes by restoring the backup.

4. **Examine Master Configuration:** The configuration file `/etc/postfix/master.cf` defines how Postfix handles different services (like SMTP). An incorrectly configured master file could prevent logging. Verify the entries in this file and compare them with defaults or known-good configurations.
5. **External Services:** If you're using external security or spam prevention services (like `fail2ban`, `SELinux`, or others), they might be affecting Postfix's operation. Check logs for these services as well.
6. **Configuration Check:** Postfix has a built-in command to check for configuration errors:

```
bash
sudo postfix check
```

This command will print out any syntax errors in your Postfix configuration files.

If none of these steps yield useful information, you might be facing a deeper system or Postfix issue. In extreme cases, you might consider reinstalling Postfix or consulting specialized forums or Postfix's official mailing lists for assistance.



```
tom@plan6:/etc/letsencrypt$ ss -tln | grep 587 tcp LISTEN 0 100 0.0.0.0:587 0.0.0.0:* tom@plan6:/etc/letsencrypt$ ss -tln | grep 587 tcp LISTEN 0 100
0.0.0.0:587 0.0.0.0:* tom@plan6:/etc/letsencrypt$ sudo systemctl restart postfix tom@plan6:/etc/letsencrypt$ sudo systemctl journal -l postfix Unknown
command verb journal. tom@plan6:/etc/letsencrypt$ tail -f /var/log/mail.log tail: cannot open '/var/log/mail.log' for reading: Permission denied tail: no files
remaining tom@plan6:/etc/letsencrypt$ tail -f /var/log/maillog tail: cannot open '/var/log/maillog' for reading: No such file or directory tail: no files remaining
tom@plan6:/etc/letsencrypt$ tail -f /var/log/syslog tail: cannot open '/var/log/syslog' for reading: Permission denied tail: no files remaining
tom@plan6:/etc/letsencrypt$ journalctl -u postfix Hint: You are currently not seeing messages from other users and the system. Users in groups 'adm',
'systemd-journal' can see all messages. Pass -q to turn off this notice. -- Journal begins at Thu 2023-08-10 17:21:50 UTC, ends at Sun 2023-09-03
21:15:30 U> -- No entries -- tom@plan6:/etc/letsencrypt$ sudo systemctl status postfix • postfix.service - Postfix Mail Transport Agent Loaded: loaded
```

-----

\_\_\_\_\_

[illegible]

=====

To resolve the issue:

**1. Backup the Configuration Files:** Before making any changes, always take a backup.

```
bash
sudo cp /etc/postfix/master.cf /etc/postfix/master.cf.bak
```

**2. Edit the master.cf File:** Open the `master.cf` file in a text editor, such as nano:

```
bash
sudo nano /etc/postfix/master.cf
```

Look for lines containing the parameters that were mentioned in the warnings (e.g., `mua_sender_restrictions`). Comment them out by placing a `#` at the beginning of each of those lines or remove them.

3. **Check the main.cf File:** While not directly mentioned in the errors, it's a good idea to check the `main.cf` file as well to ensure no unexpected parameters have been added.

```
bash
sudo nano /etc/postfix/main.cf
```

If you find those parameters in this file, that's okay since `main.cf` is where general configuration parameters belong. If they aren't set correctly or are unnecessary, you can adjust them here.

4. **Restart Postfix:** Once the changes are made, restart Postfix:

```
bash
sudo systemctl restart postfix
```

**5. Check the Configuration Again:** Run the Postfix configuration check command again to see if the warnings persist.

```
bash
sudo postfix check
```

-----

What does this mean to you: Sep 3 21:40:34 plan6 postfix/submission/smtpd[12182]: connect from ool-2f119a31.dyn.optonline.net[47.17.154.49] Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL CRAM-MD5 authentication failed: authentication failure Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: SASL authentication failure: Password verification failed Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL PLAIN authentication failed: authentication failure Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL LOGIN authentication failed: authentication failure Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: disconnect from ool-2f119a31.dyn.optonline.net[47.17.154.49] ehlo=2 starttls=1 auth=0/3 quit=1 commands=4/7



The provided log entries give information about an attempted connection to your Postfix mail server and its subsequent failure to authenticate.

Here's a breakdown:

### 1. Connection Attempt:

```
less
Sep 3 21:40:34 plan6 postfix/submission/smtpd[12182]: connect from ool-2f119a31.dyn.optonline.net[47.17.154.49]
```

This line shows that a client from the IP address 47.17.154.49 (which resolves to ool-2f119a31.dyn.optonline.net) is attempting to connect to the Postfix server's submission service (port 587).

### 2. Authentication Failures:

```
less
Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL CRAM-MD5 authentication failed: authentication failure
Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: SASL authentication failure: Password verification failed
Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL PLAIN authentication failed: authentication failure
Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL LOGIN authentication failed: authentication failure
```

These lines indicate the client attempted to authenticate using different SASL (Simple Authentication and Security Layer) mechanisms — CRAM-MD5, PLAIN, and LOGIN. All these attempts failed. The specific reason mentioned for one of the failures is "Password verification failed", which means the supplied credentials (username and password) did not match any valid account on your Postfix server.

### 3. Disconnection:

```
less
Sep 3 21:40:35 plan6 postfix/submission/smtpd[12182]: disconnect from ool-2f119a31.dyn.optonline.net[47.17.154.49] ehlo=2 starttls=1 auth=0/3 quit=1 commands=4/7
```

This line shows the client disconnected after a few SMTP commands (EHLO, STARTTLS, AUTH, QUIT). The auth=0/3 part indicates there were three authentication attempts, all of which failed.

What this means to you:

- If this connection attempt is from you or someone you know (for example, trying to set up a mail client), it's clear that the authentication details being used are incorrect. Ensure you're using the right username and password.
- If this connection is unsolicited and repeated frequently, it might be an attempt to brute-force your email server. In such cases, consider implementing additional security measures like fail2ban, which can block IPs that have multiple failed login attempts in a short time.

Lastly, always ensure you use secure and unique passwords and regularly monitor your server logs for any unusual activity.



What is an example of fake authentication details I can add to test system



If you're looking to test a system with fake authentication details, it's crucial to ensure you have permission to do so, especially if it's not your own system. Unauthorized testing can lead to severe consequences, including legal actions.

For the sake of illustration, if you want to test a system with fake credentials, you can use something clearly identifiable as fake, so there's no confusion later. Here's an example:

**Username:** testuser1234 **Password:** FakePassword!@34

Ensure these credentials don't accidentally match any real user's credentials on your system. Also, when testing systems for security, always document your actions and have a clear intent. If possible, conduct such tests in a controlled environment or a staging setup instead of a production system.

Once you're done testing, review the system logs to ensure you can identify and track these false attempts. This process can give you an idea of what a malicious or unauthorized access attempt might look like in your logs.



How do I add it

-----



If you want to add fake authentication details to test the email system (Postfix, in your case), you'd typically add a fake user to the system or to the specific authentication mechanism you're using with Postfix. Here's a general guide on how to add a fake user for testing purposes:

### 1. Using System Users (PAM authentication):

If your Postfix is set up to authenticate against system users via PAM:

- a. Add a new user to your system:

```
bash
sudo adduser testuser1234
```

- b. Set the password for the new user:

```
bash
sudo passwd testuser1234
```

You can then try to send an email using the `testuser1234` credentials.

### 2. Using Virtual Mailbox with Dovecot:

If you're using virtual mailboxes with Dovecot:

- a. Add the user to your virtual user file. This might be something like `/etc/dovecot/passwd`. Add a line like:

```
css
testuser1234:{PLAIN}FakePassword!@34
```

- b. Reload/restart Dovecot to apply changes:

```
bash
sudo systemctl restart dovecot
```

### 3. Using MySQL/PostgreSQL database for authentication:

If you're using a database for mail user authentication:

- a. Log into the database.
- b. Add a new entry to your user/mailbox table with the fake username and password.
- c. Depending on your setup, you might need to hash the password before inserting it.

Remember to remove the test user after your testing to maintain security.

Lastly, always backup configuration and user files before making any changes to them. It ensures you have a fallback if something goes wrong.



how do I add a useer with saslauthd

-----



To add a user for authentication with `saslauthd`, you would typically use the `saslpasswd2` tool. This tool creates (or updates) user entries in the SASL password database. Here's how to use it:

### 1. Install the Necessary Packages:

If you haven't installed `sasl2-bin`, install it first. On a Debian/Ubuntu system, you can do this with:

```
bash
sudo apt-get install sasl2-bin
```

### 2. Add or Update a User:

To add or update a user's password, use:

```
bash
sudo saslpasswd2 -c username
```

Replace `username` with your desired username. You'll be prompted to enter a password for the user. The `-c` option tells `saslpasswd2` to create the user if it doesn't exist.

This will add the user to the default SASL database, typically located at `/etc/sasldb2`.

### 3. Ensure `saslauthd` is Configured to Use `sasldb`:

Your `saslauthd` should be configured to use the `sasldb` mechanism. This is typically done in the `/etc/default/saslauthd` file or the main configuration for `saslauthd`.

Check for a line that looks like:

```
bash
MECHANISMS="sasldb"
```

If it's not set to `sasldb`, you'll need to modify the configuration to use it.

### 4. Restart Services:

After adding or updating a user, and ensuring `saslauthd` is configured properly, you should restart the `saslauthd` service:

```
bash
sudo systemctl restart saslauthd
```

### 5. Test the Authentication:

You can test the authentication with the `testsaslauthd` command:

```
bash
testsaslauthd -u username -p password
```

Replace `username` and `password` with the credentials you added. It should return `0: OK "Success."` if the authentication was successful.

Remember that while `sasldb` can be used for authentication, it might not scale well for larger setups. If you anticipate many users or high authentication rates, you might consider using another backend for `saslauthd`, like LDAP or SQL.

-----



You:

how can I test logins locally?

-----



Testing logins locally, especially for services like Postfix that use SASL for authentication, can be done in various ways. Here's how you can test it:

### 1. Use `testsaslauthd` to Test SASL Authentication Directly:

The `testsaslauthd` command is a utility provided by the `sasl2-bin` package to test `saslauthd` configurations. You can use it to simulate a login:

```
bash
testsaslauthd -u username -p password
```

This will query `saslauthd` directly with the provided username and password.

## 2. Use `swaks` to Test Email Authentication:

`swaks` (Swiss Army Knife for SMTP) is a powerful tool for testing SMTP servers. It's useful for sending test emails, but also for testing various parts of your mail setup, including authentication.

To install `swaks`:

```
bash
sudo apt-get install swaks
```

To test SMTP authentication:

```
bash
swaks --to test@example.com --server localhost:587 --auth LOGIN --auth-user username --auth-password password
```

Adjust the options as necessary for your setup.

## 3. Use `telnet` or `openssl` to Manually Test SMTP Authentication:

You can manually interact with your SMTP server to test authentication. This is more involved than the previous methods, but it gives you a clear step-by-step view of the SMTP authentication process.

First, start a connection to your SMTP server. If it's using TLS:

```
bash
openssl s_client -connect localhost:587 -starttls smtp
```

If not using TLS, you can use `telnet`:

```
bash
telnet localhost 587
```

Then follow the SMTP protocol to authenticate:

```
lua
EHLO test.local
AUTH LOGIN
```

The server will then reply with a base64 prompt for the username, then the password. You can convert your username and password to base64 using a tool like `base64`:

```
bash
echo -n "username" | base64
echo -n "password" | base64
```

Use the output values in your SMTP session.

Remember, always be careful when dealing with authentication, especially in a production environment. Only use tools and techniques that you trust and understand.



```
tom@plan6:/etc/letsencrypt$ swaks --to twcookson@gmail.com --server localhost:587 --auth LOGIN --auth-user test --auth-password redacted === Trying
localhost:587... === Connected to localhost. <- 220 plan6.com ESMTP Postfix (Debian/GNU) -> EHLO plan6.com <- 250-plan6.com <- 250-PIPELINING
<- 250-SIZE 10240000 <- 250-VRFY <- 250-ETRN <- 250-STARTTLS <- 250-ENHANCEDSTATUSCODES <- 250-8BITMIME <- 250-DSN <- 250-
SMTPUTF8 <- 250 CHUNKING *** Host did not advertise authentication -> QUIT <- 221 2.0.0 Bye === Connection closed with remote host.
```





The output suggests that the SMTP server (Postfix) running on localhost did not advertise any authentication mechanisms after the EHL0 command. This typically means that the Postfix server is not configured for authentication on the specified port (587 in this case) or it's missing necessary configurations.

Here are a few steps to troubleshoot:

### 1. Check master.cf Configuration:

Open the /etc/postfix/master.cf file and look for the submission section:

```
bash
nano /etc/postfix/master.cf
```

You should have something like:

```
bash
submission inet n      -      y      -      -      smtpd
  -o syslog_name=postfix/submission
  -o smtpd_tls_security_level=encrypt
  -o smtpd_sasl_auth_enable=yes
  ...
```

Ensure that -o smtpd\_sasl\_auth\_enable=yes is there and not commented out. This enables SASL authentication for the submission service.

### 2. Ensure saslauthd is Running:

```
bash
service saslauthd status
```

If it's not running, start it:

```
bash
service saslauthd start
```

### 3. Check Postfix Main Configuration:

Ensure you have the following lines in your /etc/postfix/main.cf:

```
bash
smtpd_sasl_auth_enable = yes
smtpd_sasl_security_options = noanonymous
smtpd_sasl_type = cyrus
```

If you made changes to the configuration, reload Postfix:

```
bash
service postfix reload
```

### 4. Try Connecting with STARTTLS:

Since the server advertises the STARTTLS command, it's possible that authentication is only available after the TLS handshake. You can force swaks to use STARTTLS:

```
bash
swaks --to twcookson@gmail.com --server localhost:587 --auth LOGIN --auth-user test --auth-password redacted
```

### 5. Review Logs:

Check the /var/log/mail.log for any error messages related to SASL or authentication when you try to connect.

By following these steps, you should be able to narrow down the cause of the authentication issue and remedy it.

-----



You:

```
smtp inet n - y - - smtpd -o syslog_name=postfix/submission -o smtpd_tls_security_level=encrypt -o smtpd_sasl_auth_enable=yes -o
smtpd_tls_auth_only=yes submission inet n - y - - smtpd -o syslog_name=postfix/submission -o smtpd_tls_security_level=encrypt -o
smtpd_sasl_auth_enable=yes -o smtpd_reject_unlisted_recipient=no -o smtpd_client_restrictions=$mua_client_restrictions -o
smtpd_helo_restrictions=$mua_helo_restrictions -o smtpd_sender_restrictions=$mua_sender_restrictions -o smtpd_recipient_restrictions=
smtpd_relay_restrictions=permit_sasl_authenticated,reject -o milter_macro_daemon_name=ORIGINATING
```



Your configuration for both smtp and submission services seems to be correctly set up for SASL authentication. Let's go through a checklist to further diagnose and fix the issue:

### 1. Ensure smtpd\_sasl\_path is Set Correctly:

In your /etc/postfix/main.cf, make sure you have:

```
bash
smtpd_sasl_path = private/auth
```

And the saslauthd is correctly configured to listen on that socket path.

### 2. SASL Packages:

Ensure you have the necessary SASL packages installed:

```
bash
sudo apt-get install libsasl2-2 libsasl2-modules sasl2-bin
```

### 3. SASL Configuration:

Check /etc/postfix/sasl/smtpd.conf. It should have:

```
makefile
pwcheck_method: saslauthd
mech_list: PLAIN LOGIN
```

### 4. Permissions:

Ensure postfix user has permissions to access the saslauthd socket. It's usually located at /var/spool/postfix/var/run/saslauthd/.

```
bash
chgrp postfix /var/spool/postfix/var/run/saslauthd/
```

### 5. Restart Services:

After making changes, it's always good to restart the relevant services.

```
bash
service saslauthd restart
service postfix restart
```

### 6. Review Logs:

After you try to authenticate again, check /var/log/mail.log for any error messages or warnings related to SASL or authentication. It may give more insights on what's going wrong.

### 7. Try with Telnet:

While swaks is more convenient, sometimes it's useful to manually connect using telnet:

```
bash
telnet localhost 587
```

After connecting:

```
EHLO domain.com
```

You should see a list of supported ESMTP extensions. Look for 250-AUTH PLAIN LOGIN or similar. This would confirm if authentication mechanisms are being advertised.

Remember that smtp usually listens on port 25 and is for incoming email, while submission listens on port 587 and is for client submissions. Make sure you're connecting to the right port based on what you're trying to test.

By going through these steps, you should hopefully be able to identify the issue. If all these steps do not help, you may want to consider the specific integration of the authentication backend you're using with saslauthd (like PAM, LDAP, etc.) and ensure it's correctly set up.



You:

I don't see `smtpd_sasl_path = private/auth` in there, do you? # See `/usr/share/postfix/main.cf.dist` for a commented, more complete version # Debian specific: Specifying a file name will cause the first # line of that file to be used as the name. The Debian default # is `/etc/mailname`. #myorigin = `/etc/mailname` `smtpd_banner = $myhostname ESMTP $mail_name (Debian/GNU)` `biff = no` # appending `.domain` is the MUA's job. `append_dot_mydomain = no` # Uncomment the next line to generate "delayed mail" warnings #`delay_warning_time = 4h` `readme_directory = no` # See [http://www.postfix.org/COMPATIBILITY\\_README.html](http://www.postfix.org/COMPATIBILITY_README.html) -- default to 2 on # fresh installs. `compatibility_level = 2` # TLS parameters `smtpd_tls_cert_file=/etc/ssl/certs/ssl-cert-snakeoil.pem` `smtpd_tls_key_file=/etc/ssl/private/ssl-cert-snakeoil.key` `smtpd_tls_security_level=may` `smtpd_tls_CApath=/etc/ssl/certs` `smtpd_tls_security_level=may` `smtpd_tls_session_cache_database = btree:${data_directory}/smtpd_scache` `smtpd_sasl_auth_enable = yes` `smtpd_sasl_type = cyrus` `smtpd_sasl_path = smtpd` `smtpd_sasl_security_options = noanonymous` `smtpd_sasl_local_domain = $myhostname` `smtpd_recipient_restrictions = permit_mynetworks, permit_sasl_authenticated, reject_unauth_destination` `smtpd_relay_restrictions = permit_mynetworks permit_sasl_authenticated defer_unauth>` `myhostname = plan6.com` `alias_maps = hash:/etc/aliases` `alias_database = hash:/etc/aliases` `myorigin = /etc/mailname` `mydestination = $myhostname, plan6.com, localhost.com, , localhost` `relayhost = mynetworks` `= 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128` `mailbox_size_limit = 0` `recipient_delimiter = +` `inet_interfaces = all` `inet_protocols = ipv4`



ChatGPT:

From the configuration you've provided, I can indeed see the `smtpd_sasl_path`:

```
makefile
smtpd_sasl_path = smtpd
```

This setting indicates the socket location that Postfix uses to communicate with the SASL authentication server (usually Cyrus or Dovecot). By default, for cyrus, this will look in the Postfix chroot (typically `/var/spool/postfix`) for the socket, so it would be looking for something like `/var/spool/postfix/smtpd`.

Considering the typical setup, the path should more likely be something that directs Postfix to the `saslauthd` socket.

If you're using `saslauthd` for authentication, you might want to ensure that:

1. The `saslauthd` is indeed running and has the appropriate socket in place.
2. The path in `smtpd_sasl_path` points correctly to that socket.

For `saslauthd`, a common setup might have the socket at `/var/run/saslauthd/mux`. In this case, because Postfix will be running in a chroot, you might symlink that to `/var/spool/postfix/var/run/saslauthd/`. Therefore, the correct setting for `smtpd_sasl_path` in such a setup would be:

```
makefile
smtpd_sasl_path = private/auth
```

But, this will depend on your exact setup.

To rectify:

1. Ensure `saslauthd` is running:

```
bash
service saslauthd status
```

2. If it's running, check where the socket is. This can often be found in the `saslauthd` config or by searching in `/var/run/saslauthd/`.
3. Update `smtpd_sasl_path` in `main.cf` to point to this socket, keeping in mind the Postfix chroot.
4. Restart Postfix:

```
bash
service postfix restart
```

5. Try your authentication test again.

Always ensure you're making backups of configuration files before making changes, and ensure you're following security best practices, especially when dealing with email servers and authentication!



You:

`tom@plan6:/etc/letsencrypt$ service saslauthd status -bash: service: command not found tom@plan6:/etc/letsencrypt$`



It looks like you're on a system that doesn't have the `service` command available, or it's not in your `PATH`. On newer systems, especially those using `systemd`, the `systemctl` command replaces the older `service` command for managing services.

Let's use `systemctl` to check the status of `saslauthd`:

```
bash
sudo systemctl status saslauthd
```

If `saslauthd` is not running, you can start it with:

```
bash
sudo systemctl start saslauthd
```

And to ensure it starts on boot:

```
bash
sudo systemctl enable saslauthd
```

After starting the service, you can revisit the Postfix configuration to ensure it's set up correctly to interface with `saslauthd`.



You:

```
tom@plan6:/etc/letsencrypt$ sudo systemctl status saslauthd ● saslauthd.service - LSB: saslauthd startup script
Loaded: loaded (/etc/init.d/saslauthd; generated)
Active: active (running) since Sun 2023-09-03 21:06:10 UTC; 1h 4min ago
Docs: man:systemd-sysv-generator(8)
Process: 9961 ExecStart=/etc/init.d/saslauthd start (code=exited, status=0/SUCCESS)
Tasks: 5 (limit: 465)
Memory: 1.6M
CPU: 40ms
CGroup: /system.slice/saslauthd.service ── 9997 /usr/sbin/saslauthd -a sasl_db -c -m /var/run/saslauthd -n 5 ── 9998 /usr/sbin/saslauthd -a sasl_db -c -m /var/run/saslauthd -n 5 ── 9999 /usr/sbin/saslauthd -a sasl_db -c -m /var/run/saslauthd -n 5 ── 10000 /usr/sbin/saslauthd -a sasl_db -c -m /var/run/saslauthd -n 5 ── 10001 /usr/sbin/saslauthd -a sasl_db -c -m /var/run/saslauthd -n 5
Sep 03 21:06:09 plan6.com saslauthd[9932]: Stopping SASL Authentication Daemon: sas>
Sep 03 21:06:09 plan6.com systemd[1]: saslauthd.service: Succeeded.
Sep 03 21:06:09 plan6.com systemd[1]: Stopped LSB: saslauthd startup script.
Sep 03 21:06:09 plan6.com systemd[1]: Starting LSB: saslauthd startup script...
Sep 03 21:06:10 plan6.com saslauthd[9997]: : master pid is: 9997
Sep 03 21:06:10 plan6.com saslauthd[9997]: : listening on socket: />
Sep 03 21:06:10 plan6.com saslauthd[9961]: Starting SASL Authentication Daemon: sas>
Sep 03 21:06:10 plan6.com systemd[1]: Started LSB: saslauthd startup script.
```



It appears that `saslauthd` is active and running, which is a good sign. The service is configured to use `sasl_db` for authentication.

Given the current status of `saslauthd`, let's revisit a few things:

- SASL Configuration for Postfix:** Your configuration indicates that the path to the SASL socket for postfix is `smtpd`. However, by default, `saslauthd` typically uses `/var/run/saslauthd`. You may need to adjust the `smtpd_sasl_path` in your postfix configuration.

You can try changing this line:

```
makefile
smtpd_sasl_path = smtpd
```

to:

```
javascript
smtpd_sasl_path = /var/run/saslauthd
```

- Test Authentication:** Before attempting to authenticate with Postfix, you can test the `saslauthd` authentication directly using the `testsaslauthd` command.

```
bash
testsaslauthd -u test -p redacted
```

If successful, you should see:

```
arduino
0: OK "Success."
```

If you see an error, this will give you a clue as to what might be wrong with `saslauthd` or the user database.

3. **Postfix Reload:** After making any changes to the Postfix configuration, remember to reload Postfix:

```
bash
sudo systemctl reload postfix
```

4. **Try swaks again:** Once you've verified that `saslauthd` authentication works and you've adjusted the Postfix configuration accordingly, try using `swaks` again to test email sending with authentication.



how do i install testsaslauthd

-----



To install `testsaslauthd`, you typically need to install the `sasl2-bin` package, which contains various Cyrus SASL utilities, including `testsaslauthd`.

You can install it using the following command on a Debian or Ubuntu system:

```
bash
sudo apt-get update
sudo apt-get install sasl2-bin
```

After installing, you should be able to run the `testsaslauthd` command as mentioned in the previous steps.

-----



smtpd\_sasl\_type = cyrus What are the other types, pros and cons

-----



The `smtpd_sasl_type` parameter in Postfix specifies which SASL (Simple Authentication and Security Layer) implementation to use for authentication. The primary options available for this parameter are:

1. **cyrus:** This refers to the Cyrus SASL library, which is a widely-used, general-purpose authentication library.
2. **dovecot:** This uses the authentication mechanisms provided by the Dovecot IMAP/POP3 server.

#### Pros and Cons:

##### 1. Cyrus SASL:

###### Pros:

- General-purpose: It's a standalone library, designed to be used by any application that needs SASL functionality.
- Versatile: Supports various authentication methods, including PLAIN, CRAM-MD5, and DIGEST-MD5.
- Well-established: It's a well-known library with a long history.

###### Cons:

- Complexity: Might be overkill for simpler setups or when only basic authentication mechanisms are needed.
- Additional management: Requires a separate management for its user database unless integrated with another system (like PAM).

##### 2. Dovecot SASL:

###### Pros:

- Integrated: If you're already using Dovecot for IMAP/POP3, it provides a seamless integration, allowing you to have one authentication source for both email retrieval and sending.
- Simplicity: Dovecot's configuration can be easier to grasp for newcomers, especially in simpler setups.
- Performance: Dovecot is optimized for performance and can handle a lot of simultaneous connections.

Cons:

- Specificity: It's primarily meant for use with the Dovecot server. If you're not using Dovecot for anything else, it might not make sense to use Dovecot's SASL for Postfix.

## Which to choose?

The best choice largely depends on your setup:

- If you're already using Dovecot for IMAP/POP3, it makes sense to use Dovecot SASL for simplicity and integrated management.
- If you're not using Dovecot, or if you need some of the advanced features of the Cyrus SASL library, then Cyrus SASL might be a better choice.

Always consider your specific needs, the scale of your mail system, and your comfort level with each option.



/etc/postfix/sasl/smtpd.conf does not exist

-----



The absence of /etc/postfix/sasl/smtpd.conf indicates that the Cyrus SASL configuration file for Postfix hasn't been created. If you've chosen to use Cyrus SASL for authentication in Postfix (as indicated by smtpd\_sasl\_type = cyrus), then you'll need a configuration for it.

Here's a simple way to create a basic smtpd.conf for Cyrus SASL:

1. **Create the directory** if it doesn't exist:

```
bash
sudo mkdir -p /etc/postfix/sasl
```

2. **Create and edit the configuration file:**

```
bash
sudo nano /etc/postfix/sasl/smtpd.conf
```

3. **Add the following basic configuration:**

```
makefile
pwcheck_method: saslauthd
mech_list: PLAIN LOGIN
```

This configures Cyrus SASL to use the saslauthd daemon for authentication and enables the PLAIN and LOGIN mechanisms.

4. **Save and exit the editor.** If using nano, press CTRL + O to save and CTRL + X to exit.

5. **Ensure permissions are correct:**

```
bash
sudo chown -R postfix:postfix /etc/postfix/sasl/
sudo chmod 750 /etc/postfix/sasl/
```

6. **Reload Postfix** to apply changes:

```
bash
sudo systemctl reload postfix
```

7. **Test your configuration** to ensure SASL authentication is working as expected.

Always make sure to validate any changes in a test environment (if possible) before applying them to a production server. Adjust the configuration based on your specific needs and security requirements.

-----



You:

Sep 3 22:23:06 plan6 postfix/master[13176]: reload -- version 3.5.18, configuration /etc/postfix  
Sep 3 22:23:26 plan6 postfix/submission/smtpd[13293]: disconnect from plan6.com[127.0.0.1] ehlo=1 quit=1  
commands=2  
Sep 3 22:23:56 plan6 postfix/submission/smtpd[13296]: connect from ool-2f119a31.dyn.optonline.net[47.17.154.49]  
Sep 3 22:23:56 plan6 postfix/submission/smtpd[13296]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL CRAM-MD5 authentication failed: authentication failure  
Sep 3 22:23:56 plan6 postfix/submission/smtpd[13296]: warning: SASL authentication failure: Password verification failed  
Sep 3 22:23:56 plan6 postfix/submission/smtpd[13296]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL PLAIN authentication failed: authentication failure  
Sep 3 22:23:56 plan6 postfix/submission/smtpd[13296]: warning: ool-2f119a31.dyn.optonline.net[47.17.154.49]: SASL LOGIN authentication failed: authentication failure  
Sep 3 22:23:56 plan6 postfix/submission/smtpd[13296]: disconnect from ool-2f119a31.dyn.optonline.net[47.17.154.49] ehlo=2  
starttls=1 auth=0/3 quit=1 commands=4/7



ChatGPT:

From the log provided:

1. The Postfix service has been reloaded, likely due to configuration changes.
2. There's a local connection from plan6.com (127.0.0.1). It looks like something on your server briefly connected to Postfix and then disconnected without attempting any real communication (just EHLO and QUIT commands). This might be a simple connectivity check or some software just making sure the SMTP service is alive.
3. An external connection is made from ool-2f119a31.dyn.optonline.net (with IP address 47.17.154.49). It tries three different authentication mechanisms (CRAM-MD5, PLAIN, and LOGIN), and all of them fail. After these failures, the client disconnects.

Key takeaways and things to consider:

- **Repeated Authentication Failures:** The authentication failures could indicate a few things:
  - A legitimate user is having trouble remembering or entering their password correctly.
  - Someone is trying to brute force your SMTP authentication.
  - Misconfiguration in a mail client, leading to repeated authentication failures.
- **SASL CRAM-MD5:** This method is an older and less commonly used mechanism. If you aren't expecting any users to authenticate with CRAM-MD5, you might consider disabling it to reduce the potential attack surface. However, before you do this, you should ensure no legitimate users are reliant on this mechanism.
- **Monitoring & Rate Limiting:** If you see a lot of repeated authentication failures from the same IP (especially in a short period), you might consider implementing fail2ban or similar tools to temporarily block IPs that have too many failed authentication attempts in a set time frame. This can help prevent potential brute force attacks.
- **Check Authentications:** Try logging in using a known good user/password combination to ensure that your configuration is correct and that SASL authentication is working as expected.

If the authentication failures are from your attempts, it could be due to incorrect settings in the email client or issues with the Cyrus SASL setup. In that case, you would want to double-check the configuration and the client's settings. If these are not your attempts, it's crucial to monitor these events and ensure that your mail server is secured against unauthorized access.



You:

Yeah, that was me trying to send an email wiht a known good un and pw



ChatGPT:

Alright, since it was you trying to send the email and the authentication failed, there are a few things we should look into:

1. **SASL Configuration:** Your Postfix is set up to use cyrus as the SASL type. Ensure that Cyrus SASL is correctly set up and can authenticate against the database or authentication source you've chosen. Since the smtpd.conf for SASL configuration was missing earlier, it's essential to ensure that the correct mechanisms and paths are defined there.
2. **Postfix Configuration:** Double-check the smtpd\_sasl\_path in your Postfix configuration. By default, for Cyrus SASL, it might be looking for a socket in /var/run/saslauthd/, but this can vary based on setup.
3. **Testing SASL Directly:** You can use testsaslauthd to directly check authentication:

```
bash
testsaslauthd -u username -p password
```

If this fails, then there's a problem with saslauthd configuration or the underlying authentication mechanism it's using.

4. **Ensure Password Database:** If you're using sasl2db for Cyrus SASL, ensure that the users are added using `saslpasswd2`.
5. **Client Configuration:** Ensure that the email client is set up correctly:
  - Correct username and password.
  - Using the right authentication method. If the client is set to use CRAM-MD5 and the server doesn't support it, for instance, there will be a mismatch.
6. **Postfix Logging:** Increase the verbosity of the logging for Postfix to get more details on where things might be going wrong. This can be done by setting `debug_peer_list` in your `main.cf`:

```
bash
debug_peer_list = 127.0.0.1, [external IP]
```

After making any changes, remember to reload or restart the Postfix and Cyrus SASL services to ensure the new settings take effect.

Investigating mail server issues can be complex due to the myriad of configurations and interactions between different components. Working step-by-step, verifying each component, and ensuring detailed logging can help in identifying and resolving the problem.

-----



You:

lets remove postfix and related stuff I'm going to try something different

-----



ChatGPT:

Sure, if you're thinking of starting afresh or trying a different mail server solution, removing Postfix and related components is a good step. Here's how you can remove Postfix and its related packages on a Debian-based system:

1. **Stop the services:**

```
bash
sudo systemctl stop postfix
sudo systemctl stop saslauthd
```

2. **Remove Postfix and related packages:**

```
bash
sudo apt-get purge postfix* sasl2-bin libsasl2-modules
```

3. **Remove configuration and related directories:**

```
bash
sudo rm -rf /etc/postfix/
sudo rm -rf /etc/sasl2/
sudo rm -rf /var/spool/postfix/
```

4. **Optionally, remove related users and groups:**

```
bash
sudo deluser postfix
sudo delgroup postfix
```

5. **Update the package database:**

```
bash
sudo apt-get update
```

6. **Optionally, run autoremove to get rid of any orphaned packages:**

```
bash
sudo apt-get autoremove
```

After completing these steps, Postfix and its related components should be removed from your system. If you're thinking of trying a different mail server solution, there are several options available, such as Exim, Sendmail, Dovecot (primarily for IMAP/POP3), etc.



When setting up a new mail server, it might be helpful to start with a clear plan and list of requirements, as this will guide the configuration process and ensure that all necessary security and functionality measures are in place.



Do you know anything about esmtp?



Yes, I'm familiar with ESMTP.

ESMTP stands for Extended Simple Mail Transfer Protocol. It is an extension to the Simple Mail Transfer Protocol (SMTP), which is the core protocol used for sending emails on the Internet. While the original SMTP had a fixed set of commands, ESMTP allows for a more flexible set through the use of extensions.

Some of the features that ESMTP brought to the table over standard SMTP include:

1. **Authentication:** The ability for the client to authenticate to the server, which is critical for security and spam prevention. This typically uses mechanisms like PLAIN, LOGIN, or CRAM-MD5 via the AUTH command.
2. **StartTLS:** It supports initiating a TLS (Transport Layer Security) session to encrypt the entire SMTP conversation. This ensures the confidentiality and integrity of the transmitted data.
3. **Enhanced status codes:** These provide more detailed information about the success or failure of message delivery.
4. **Chunking:** Allows for sending the message in chunks, which can be beneficial for large emails.
5. **Binary MIME:** ESMTP can transmit emails that have binary data without the need to encode them as ASCII text first.
6. **Delivery status notifications:** Gives detailed delivery reports back to the sender.
7. **Pipelining:** Improves performance by sending multiple commands at once without waiting for a response after each one.

There's also a software called "esmtp" which is a user-configurable relay-only Mail Transfer Agent (MTA) with a sendmail-compatible syntax. It's designed to be used by programs that need to send email, without the overhead and complexity of running and maintaining a full-fledged MTA.

If you're considering using the "esmtp" software or if you have questions about the ESMTP protocol extensions, let me know how I can assist you further!



Lets install what you were refering to here: There's also a software called "esmtp" which is a user-configurable relay-only Mail Transfer Agent (MTA) with a sendmail-compatible syntax. It's designed to be used by programs that need to send email, without the overhead and complexity of running and maintaining a full-fledged MTA.



Certainly. Installing "esmtp" is relatively straightforward. Here's a general guide, though the exact steps might vary depending on your specific Linux distribution:

### Installing esmtp on Debian-based systems (e.g., Ubuntu):

1. **Update package lists:**

```
bash
sudo apt update
```

2. **Install esmtp:**

```
bash
sudo apt install esmtp
```

### Installing esmtp on Red Hat-based systems (e.g., Fedora, CentOS):

1. **Update package lists:**

```
bash
sudo yum update
```

## 2. Install esmtp:

```
bash
sudo yum install esmtp
```

(Note: Depending on the exact distribution and its version, the package might or might not be available in the default repositories.)

### Basic Configuration:

After installation, you'll need to configure esmtp. The configuration file is usually located at `~/.esmtp.rc` for user-specific configurations or `/etc/esmtp.rc` for system-wide configurations.

Here's a basic example:

```
conf
identity "default"
hostname smtp.yourmailprovider.com:587
username "yourusername"
password "yourpassword"
starttls enabled
```

Replace `smtp.yourmailprovider.com`, `yourusername`, and `yourpassword` with your SMTP server details.

### Test:

You can then use esmtp to send an email:

```
bash
echo -e "Subject: Test Mail\n\nThis is a test mail." | esmtp recipient@example.com
```

Replace `recipient@example.com` with the email address you want to send the test mail to.

Do remember that using esmtp means your application or script will need to format the email correctly (including headers) before passing it to esmtp for sending.

-----



You:

How do I restart the process with my updated config and send a test mail with my new username and password

-----



ChatGPT:

Once you've updated the esmtp configuration file, you don't need to restart any service or process, as esmtp doesn't run as a daemon. It executes on demand when you invoke it to send an email. This is different from full-fledged MTAs like Postfix or Sendmail that run as services.

Here's how you can send a test email using esmtp:

#### 1. Write the Email Content:

Create a simple text file named `testmail.txt`:

```
bash
nano testmail.txt
```

Then, insert the following content:

```
vbnet
To: recipient@example.com
From: your-email@example.com
Subject: Test Email

This is a test email sent from esmtp.
```

Make sure to replace `recipient@example.com` with the actual recipient's email address and `your-email@example.com` with your email address.

#### 2. Send the Email:

Use the `esmtplib` command to send the email:

```
bash
esmtplib -t < testmail.txt
```

The `-t` option indicates that the recipient (and other addressing details) should be taken from the headers in the email content.

After executing the above command, check the recipient's inbox (and maybe the spam folder) to see if the email was delivered.

Remember, for `esmtplib` to work correctly, the `~/.esmtplib` or `/etc/esmtplib` configuration file must be set up with the correct SMTP server details, including the username and password.



It doesn't need the password unless it's relaying?



The need for a password in `esmtplib` (or any SMTP client for that matter) is based on the SMTP server's configuration, specifically its authentication requirements.

- 1. Direct Sending to the Recipient's SMTP Server (Not Common):** If you're trying to send an email directly to the recipient's SMTP server (without using a relay SMTP server like Gmail, Sendgrid, etc.), then you typically don't need a password. But this method is less common and less reliable because many email servers on the internet won't accept direct connections from unknown servers, especially home IPs, due to spam concerns.
- 2. Using a Relay SMTP Server (Common):** Most users send emails using a relay SMTP server (e.g., your email provider's SMTP server, such as Gmail's SMTP, Sendgrid, etc.). This is the most common setup. When you're sending through a relay SMTP server, you usually need to authenticate with a username and password. The relay server then takes care of delivering the email to the final destination.

For `esmtplib`, you will typically set it up with the SMTP details of a relay server, and in most cases, this will require authentication. So, you'd often provide a username and password in the configuration (`~/.esmtplib` or `/etc/esmtplib`).

If you're using a well-known email provider's SMTP server or a professional SMTP relay service, you will almost certainly need to provide authentication credentials. If you have your SMTP server, the need for a password will depend on how you've configured it.



How do I send a message for `esmtplib` to send from another computer with a username and password?



To send an email with `esmtplib` from another computer, using a username and password for SMTP authentication, follow these steps:

- 1. Installation:** If `esmtplib` is not installed, first install it on that computer. The method to install will depend on your distribution, but on Debian-based systems, you can do:

```
arduino
sudo apt-get install esmtplib
```

- 2. Configuration:** Create a configuration file for `esmtplib`. This can be a per-user configuration (`~/.esmtplib`) or a system-wide configuration (`/etc/esmtplib`).

Here's an example of what the file might contain:

```
conf
hostname=smtp.your-email-provider.com:port
username=your-email@example.com
password=your-password
starttls=enabled
```

Replace the placeholders with appropriate values for your SMTP server, your email, and your password.

3. **Send an Email:** Create a test email in a file, for example `test-email.txt`:

```
vbnet
To: recipient@example.com
From: your-email@example.com
Subject: Test ESMTP Mail

This is a test email sent using ESMTP.
```

Use `esmtplib` to send the email:

```
esmtplib -t < test-email.txt
```

The `-t` option tells `esmtplib` to extract recipients from the email headers.

Remember, the security of transmitting username/password combinations in plain text or storing them in configuration files is questionable. Ensure you protect the `.esmtplib` file:

```
bash
chmod 600 ~/.esmtplib
```

This ensures that only the file's owner can read or write to it, which provides some degree of protection against unintentional exposure.

However, the best practice would be to use more secure methods, like OAuth2 for SMTP, but configuring that would be more complex and may not be directly supported by simple tools like `esmtplib`. Always keep security in mind when dealing with email sending and authentication.



from another computer

-----



To send an email from another computer using `esmtplib`, you'll need to:

1. **Install esmtplib:** Install `esmtplib` on the other computer. This will vary depending on the OS of that computer.
2. **Configuration:** On the other computer, configure `esmtplib` the same way we did before. Set up an `.esmtplib` file in the user's home directory or in `/etc/`.
3. **Craft the Email:** On the other computer, create a file (`test-email.txt`, for example) with your email contents:

```
vbnet
To: recipient@example.com
From: your-email@example.com
Subject: Test ESMTP Mail

This is a test email sent from another computer using ESMTP.
```

4. **Send the Email:** Run the following command on the other computer:

```
esmtplib -t < test-email.txt
```

If the other computer is on a different network, ensure that it can reach your SMTP server (the one `esmtplib` is configured to relay through). Also, check if the SMTP server allows connections from different IP addresses using the same credentials. Some email providers might block unfamiliar login locations for security reasons.

If you're facing issues, the error messages (or log entries, if `esmtplib` provides them) would be the next point of investigation. They will give a clue about what might be wrong, such as authentication failures, unreachable servers, etc.



But I mean a different computer from the one `esmtplib` is installed on. How do I relay an email from another computer

-----



Ah, I understand now. You wish to send an email from a different computer and have it relayed through the computer on which esmtplib is installed.

To achieve this, you'll need an SMTP client on the other computer and then set up the computer with esmtplib as an SMTP relay. Here's a step-by-step guide:

1. **SMTP Client Installation:** Install an SMTP client on the other computer. This can be a mail client like Thunderbird, Outlook, or even command-line tools like swaks.
2. **SMTP Client Configuration:** Configure the SMTP client to use the esmtplib computer as its SMTP server:
  - SMTP server: [IP address or hostname of the computer where esmtplib is installed]
  - Port: Typically 25, 587, or whichever port you've set esmtplib to listen on.
  - Username: [your username]
  - Password: [your password]
  - Encryption: Depending on your esmtplib setup, this could be none, SSL, or TLS.
3. **ESMTPLIB Configuration:** Ensure that esmtplib is configured to listen for incoming connections and is set up to relay mail. This often involves configuring it to listen on a network-accessible IP (not just localhost) and ensuring that appropriate security mechanisms are in place.

Make sure you have proper authentication mechanisms in place to prevent unauthorized use of your relay.
4. **Firewall & Networking:** Ensure that the firewall on the esmtplib computer allows incoming connections on the SMTP port. If both computers are on different networks, ensure that the necessary ports are forwarded if there's a router or firewall in between.
5. **Send Test Mail:** Now, from the other computer, try sending an email using the SMTP client. It should connect to the esmtplib server, authenticate, and then esmtplib will relay the email to its final destination.

Note: Running a mail relay can expose you to various risks, especially if it's misconfigured. Open relays can be abused by spammers. Always ensure that you have proper authentication in place and regularly monitor the traffic to avoid misuse.



You:

Does it need help from another program to listen for a port?



No, esmtplib (or any typical MTA or SMTP relay software) doesn't require an additional program to listen on a port. When properly configured, it will bind to and listen on the specified port by itself.

However, there are some important considerations:

1. **Port Permissions:** In Unix-like operating systems, binding to ports below 1024 requires root privileges. Therefore, if you're trying to use a low-numbered port (like the default SMTP port 25), the program might need elevated privileges to start.
2. **Firewall:** If there's a firewall running on the machine (like iptables, ufw, etc. on Linux), it may block external connections to the port esmtplib is listening on. You'd need to configure the firewall to allow incoming connections on that port.
3. **Address Binding:** The program should be configured to bind to an external-facing IP address or 0.0.0.0 (all available interfaces) if you want to access it from another computer. If it binds to 127.0.0.1 (localhost), it will only be accessible from the same machine.
4. **Other Services:** Make sure no other service is listening on the same port that esmtplib is configured to use. For example, if you had Postfix running and listening on port 25, you'd have a conflict if esmtplib tried to use the same port.

While esmtplib doesn't require another program to listen on a port, you might want to use a service supervisor (like systemd, supervisord, etc.) to manage its process, ensuring it starts at boot and restarts if it crashes. But that's more about process management than port listening.



You:

How do I check if esmtplib is listening?



You can use the `netstat` or `ss` commands to check if a program is listening on a specific port. To determine if `esmtplib` is listening:

### 1. Using `netstat`:

If you don't have `netstat` installed, on a Debian-based system, you can get it by installing the `net-tools` package:

```
arduino
sudo apt-get install net-tools
```

Then, run:

```
perl
netstat -tuln | grep esmtplib
```

If `esmtplib` is listening on its default port (or whichever port you configured), you should see an entry in the results.

### 2. Using `ss`:

The `ss` command is a utility to investigate sockets, and it's a replacement for `netstat`. To check if `esmtplib` is listening, run:

```
perl
ss -tuln | grep esmtplib
```

### 3. Using `lsof`:

`lsof` (List Open Files) is a command to list open files, but it can also be used to see listening ports:

```
css
lsof -i | grep esmtplib
```

One of these methods should show you if `esmtplib` is actively listening on a port. If you don't see any output from the above commands, it suggests `esmtplib` isn't currently listening.

-----



tom@plan6:~\$ netstat -tuln | grep esmtplib tom@plan6:~\$ It's not listening although I installed it and its configured

-----



Alright, if `esmtplib` isn't listening, it means it's not set up as a daemon to continuously listen for incoming connections, or it hasn't been started in a mode where it operates as a server. Typically, `esmtplib` operates as a client, not as a server. It's designed to send mail from the machine it's running on, rather than to accept mail from external sources.

When you want to send email with `esmtplib`, you invoke it (either directly or through another program) to send mail. You don't connect to it like you would with a regular SMTP server.

If you're trying to set up a mail server that will listen for incoming SMTP connections, `esmtplib` may not be the tool you're looking for. Instead, you would typically use something like Postfix, Exim, or Sendmail.

However, if you're trying to relay mail from another machine through the machine with `esmtplib` installed, you would typically:

1. Have a full-fledged MTA (like Postfix) set up on the "other" machine.
2. Configure this MTA to relay mail through the machine with `esmtplib`.
3. When mail is sent from the "other" machine, its MTA would connect to the machine with `esmtplib` and relay the mail through it.

But with just `esmtplib`, you won't have a listening SMTP server that external machines can connect to directly. If you need a listening SMTP server, you'd want to go back to a more robust MTA or look into other options that provide SMTP server functionality.

-----



How do I remove esmtplib

-----



To remove esmtplib on a Debian-based system like Ubuntu, you can use the apt-get or apt command. Here's how:

```
bash
sudo apt-get remove --purge esmtplib
```

or using apt:

```
bash
sudo apt remove --purge esmtplib
```

This will uninstall esmtplib and also purge its configuration files.

Afterwards, you may also want to run:

```
bash
sudo apt-get autoremove
```

or

```
bash
sudo apt autoremove
```

This will remove any dependencies that were installed with esmtplib but are no longer used by any other package.



You:

What's a smarthost in the context of email



A smarthost is a third-party email server that acts as an intermediary relay for outbound emails from a particular sender. In the context of email, here's how a smarthost works:

1. **Relay Control:** Instead of sending email directly to the recipient's mail server, the sending server forwards the email to the smarthost. The smarthost then takes responsibility for delivering that email to the recipient's server.
2. **Use Cases:**
  - **Bulk Mailers:** Organizations that send large volumes of email (e.g., newsletters or promotional emails) may use a smarthost to offload the task of delivery, ensuring better deliverability rates.
  - **Avoiding IP Reputation Issues:** If a server's IP has a poor reputation or is on a blacklist, using a smarthost can help because the smarthost's IP address is used for the final delivery.
  - **Managing Outbound Filtering:** Smarthosts often provide features like virus scanning, content filtering, or ensuring compliance with various regulations for outbound email.
  - **Dynamic IP or Unreliable Internet Connections:** Home users or small businesses might not have a fixed IP address or a reliable internet connection suitable for direct email sending. In these cases, using a smarthost (like an ISP's mail server) ensures the mail is relayed properly.
3. **Pros:**
  - **Increased Deliverability:** Smarthosts usually have mechanisms in place to manage their IP reputation, ensuring better deliverability of emails.
  - **Scalability:** Smarthosts can handle large volumes of email, reducing the load on the original sending server.
  - **Added Security and Features:** Many smarthosts provide added security features, such as outbound filtering, ensuring no malware or spam gets sent from your network.
4. **Cons:**
  - **Dependency:** There's a dependency on a third-party server, which might be a single point of failure unless redundant smarthosts are used.
  - **Cost:** Many commercial smarthost services come with associated costs.

In many email configurations, especially in enterprise environments, a smarthost is specified in the Mail Transfer Agent (MTA) settings, like Postfix or Microsoft Exchange, to relay all outgoing emails.



You:

What does it mean for email to be sent by a smarthost



ChatGPT:

When email is sent "by a smarthost", it means that the outgoing email, rather than being sent directly from the sender's server to the recipient's server, is relayed through an intermediate server — the smarthost.

Here's a breakdown of the process:

1. **Initiation:** An email client or application creates an email and sends it to its local Mail Transfer Agent (MTA) for delivery.
2. **Relay to Smarthost:** Instead of the local MTA trying to deliver the email directly to the recipient's MTA (which could be anywhere on the internet), it sends the email to a pre-defined smarthost.
3. **Smarthost Handling:** The smarthost takes on the responsibility of delivering that email to the destination MTA. It will handle DNS lookups for the destination domain's MX records, establish connections, manage queues if the remote server is unavailable, etc.
4. **Final Delivery:** Once the smarthost has successfully delivered the email to the recipient's MTA, the email then gets passed to the recipient's Mail Delivery Agent (MDA) and then to their mailbox.

Using a smarthost can offer several advantages:

- **Improved Deliverability:** A smarthost often has better IP reputation management and can increase the chances of the email landing in the recipient's inbox rather than their spam folder.
- **Load Offloading:** For organizations sending large volumes of emails, handing off delivery to a smarthost can reduce the load on the local server.
- **Outbound Filtering:** Some smarthosts provide security features like scanning outbound emails for viruses, malware, or signs of phishing.
- **Retry Mechanisms:** If the recipient's server is temporarily unavailable, a smarthost will queue the message and keep retrying delivery at regular intervals.

To sum up, when an email is sent "by a smarthost", it's like handing off a package to a courier service. The sender trusts the smarthost to handle all the complexities of delivery, ensuring the package (or email) reaches its final destination.



You:

So I have a Django webapp that wants to send automated emails. Django will send emails for this server to send. What option is that: | General type of mail configuration: | | | internet site; mail is sent and received directly using SMTP | | mail sent by smarthost; received via SMTP or fetchmail | | mail sent by smarthost; no local mail | | local delivery only; not on a network | | no configuration at this time | | |



ChatGPT:

For a Django webapp that wants to send automated emails, the most common setup is to send mail through a smarthost (especially if you're using services like SendGrid, Mailgun, or AWS SES).

In the options you've provided, the most suitable for this scenario would be:

**mail sent by smarthost; no local mail**

This means:

- All outgoing emails (from your Django app) will be relayed through a smarthost (the designated email relay service).
- The server won't handle or process incoming emails, and there won't be local mail processing. This keeps things simple and is often ideal for web applications that only need to send out notifications or other automated emails.

Remember to configure Django with the appropriate SMTP settings to use the smarthost for email sending, and on the server side, you'll configure the smarthost's SMTP details so that emails can be relayed correctly.



You:



All outgoing emails (from your Django app) will be relayed through a smarthost (the designated email relay service). But I want this server to actually send the emails



Got it! If you want the server on which your Django app is running to directly send the emails without relaying them through an external service (smarthost), then you'll want to select:

### internet site; mail is sent and received directly using SMTP

In this configuration:

- Your server will act as its own mail server and will directly send emails to the internet using SMTP.
- It also has the capability to directly receive emails. This is often more complex to set up correctly due to concerns with deliverability, SPF/DKIM/DMARC settings, IP reputation, and more.

Remember, when you select this configuration:

1. Ensure your server's IP address isn't blacklisted. Some cloud providers' IPs can be blacklisted by default because they have been abused by spammers in the past.
2. Set up SPF, DKIM, and possibly DMARC records in your DNS settings to improve email deliverability.
3. Make sure you have proper reverse DNS (PTR) records set for your server.

After the configuration, you'll provide Django with the SMTP settings of your local server (typically localhost as the SMTP server, with the appropriate port, usually 25).

However, I would advise being careful with this setup. It's easy to end up in a situation where your emails are flagged as spam or not delivered at all. Using a dedicated relay service or smarthost like SendGrid, Mailgun, or AWS SES can alleviate many of these deliverability concerns. But if you're aware of the potential pitfalls and are prepared to manage them, then it's entirely feasible to run your own mail server.



You:

I just followed this tutorial: Install and configure Exim4  
Install Exim4 To install Exim4, run the following command: `sudo apt install exim4`  
Configure Exim4 To configure Exim4, run the following command: `sudo dpkg-reconfigure exim4-config` This displays a "wizard" user interface for configuring the software. For example, in Exim4 the configuration files are split amongst multiple files by default; if you wish to have them in one file you can configure this via the user interface. All configurable parameters from the user interface are stored in the `/etc/exim4/update-exim4.conf.conf` file. To re-configure the software you can either re-run the wizard, or manually edit this file using your preferred editor. Once you are finished, you can run the following command to generate the master configuration file: `sudo update-exim4.conf` The master configuration file is stored in `/var/lib/exim4/config.autogenerated`. Warning: You should never manually edit the master configuration file, `/var/lib/exim4/config.autogenerated`, because it is updated automatically every time you run `update-exim4.conf`. Any changes you make to this file will be lost during future updates.  
Start the Exim4 daemon The following command will start the Exim4 daemon: `sudo service exim4 start`  
SMTP authentication Exim4 can be configured to use SMTP-AUTH with Transport Layer Security (TLS) and Simple Authentication and Security Layer (SASL). First, enter the following into a terminal prompt to create a certificate for use with TLS: `sudo /usr/share/doc/exim4-base/examples/exim-gencert`  
Configure Exim4 for TLS by editing the `/etc/exim4/conf.d/main/03_exim4-config_tlsoptions` file and adding the following: `MAIN_TLS_ENABLE = yes`  
Next, configure Exim4 to use the `saslauthd` daemon for authentication by editing `/etc/exim4/conf.d/auth/30_exim4-config_examples` – uncomment the `plain_saslauthd_server` and `login_saslauthd_server` sections:  
`plain_saslauthd_server: driver = plaintext public_name = PLAIN server_condition = ${if saslauthd}{{${auth2}}{{${auth3}}}{{1}}{{0}}} server_set_id = $auth2 server_prompts = : .ifndef AUTH_SERVER_ALLOW_NOTLS_PASSWORDS server_advertise_condition = ${if eq{{${tls_cipher}}}{{*}}} .endif login_saslauthd_server: driver = plaintext public_name = LOGIN server_prompts = "Username:: : Password::" # don't send system passwords over unencrypted connections server_condition = ${if saslauthd}{{${auth1}}{{${auth2}}}{{1}}{{0}}} server_set_id = $auth1 .ifndef AUTH_SERVER_ALLOW_NOTLS_PASSWORDS server_advertise_condition = ${if eq{{${tls_cipher}}}{{*}}} .endif`  
To enable outside mail clients to connect to the new server, a new user needs to be added into Exim4 by using the following commands: `sudo /usr/share/doc/exim4-base/examples/exim-adduser`  
Protect the new password files with the following commands: `sudo chown root:Debian-exim /etc/exim4/passwd sudo chmod 640 /etc/exim4/passwd`  
Finally, update the Exim4 configuration and restart the service: `sudo update-exim4.conf sudo systemctl restart exim4.service`  
Configure SASL To configure `saslauthd` to provide authentication for Exim4, first install the `sasl2-bin` package by running this command at a terminal prompt: `sudo apt install sasl2-bin`  
To configure `saslauthd`, edit the `/etc/default/saslauthd` configuration file and set: `START=yes`  
Next, to make Exim4 use the `saslauthd` service, the Debian-exim user needs to be part of the `sasl` group: `sudo adduser Debian-exim sasl`  
Finally, start the `saslauthd` service: `sudo service saslauthd start`  
Exim4 is now configured with SMTP-AUTH using TLS and SASL authentication. How do I test it?



To test your Exim4 setup with SMTP-AUTH using TLS and SASL authentication, follow these steps:

#### 1. Check Services Status:

Before testing, ensure that both `exim4` and `saslauthd` services are running:

```
bash
```

```
sudo systemctl status exim4
sudo systemctl status saslauthd
```

## 2. Test Local Mail Delivery:

First, let's ensure that Exim can send email locally:

```
bash
echo "This is a test email" | mail -s "Test Subject" yourusername@localhost
```

Then, check your local mailbox:

```
bash
cat /var/mail/yourusername
```

You should see the test email you sent.

## 3. Use a Mail Client for Testing SMTP-AUTH:

To test the authentication and TLS features, you'll need a mail client like Thunderbird, Microsoft Outlook, or even swaks (a command-line SMTP testing tool).

Here's an example using swaks:

First, install swaks:

```
bash
sudo apt install swaks
```

Then, use it to test:

```
bash
swaks --to destination_email@example.com --from yourusername@yourserverdomain.com --server localhost:25 --aut
```

Replace `destination_email@example.com`, `yourusername@yourserverdomain.com`, `yourusername`, and `yourpassword` with appropriate values.

If everything is set up correctly, you should be able to send an email with swaks using your SASL credentials.

## 4. Check the Logs:

After sending the test email, check the Exim logs to ensure that everything worked correctly:

```
bash
sudo cat /var/log/exim4/mainlog
```

This will show you a detailed log of what Exim did when processing the email. Look for any errors or warning messages.

## 5. External Tests:

If you have a remote machine or another system outside of your network, try setting up an email client there with the SMTP settings for your Exim server. This will allow you to test how Exim behaves from external connections.

Remember, configuring and running a mail server is a complex task with various components involved. It's essential to monitor and manage the server, keeping an eye on logs, ensuring mail deliverability, and handling potential issues such as IP blacklisting or security vulnerabilities.



Does this look okay: tom@plan6:~\$ sudo systemctl status exim4 ● exim4.service - LSB: exim Mail Transport Agent Loaded: loaded (/etc/init.d/exim4; generated) Active: active (running) since Sun 2023-09-03 23:28:22 UTC; 3min 9s ago Docs: man:systemd-sysv-generator(8) Tasks: 1 (limit: 465) Memory: 484.0K CPU: 141ms CGroup: /system.slice/exim4.service └─16733 /usr/sbin/exim4 -bd -q30m Sep 03 23:28:21 plan6.com systemd[1]: exim4.service: Succeeded. Sep 03 23:28:21 plan6.com systemd[1]: Stopped LSB: exim Mail Transport Agent. Sep 03 23:28:21 plan6.com systemd[1]: Starting LSB: exim Mail Transport Agent... Sep 03 23:28:22 plan6.com exim4[16496]: Starting MTA: exim4. Sep 03 23:28:22 plan6.com systemd[1]: Started LSB: exim Mail Transport Agent. tom@plan6:~\$ sudo systemctl status saslauthd ● saslauthd.service - LSB: saslauthd startup script Loaded: loaded (/etc/init.d/saslauthd; generated) Active: active (exited) since Sun 2023-09-03 23:28:49 UTC; 2min 51s ago Docs: man:systemd-sysv-generator(8) Process: 16855 ExecStart=/etc/init.d/saslauthd start (code=exited, status=0/SUC> CPU: 13ms

-----

The status for `exim4` looks good – it's running as expected.

However, for `saslauthd`, the status is a bit ambiguous. The `active (exited)` status indicates that while the startup process completed successfully, the daemon itself isn't running in the background. This isn't the expected behavior for `saslauthd`, as it should be actively listening and processing authentication requests.

Here's what you can do:

1. **Review the logs:** Before taking any action, look into the logs for `saslauthd` to see if there are any error messages or indicators as to why it's not staying active.

```
bash
sudo journalctl -u saslauthd
```

2. **Try restarting `saslauthd`:** Sometimes, a simple restart can resolve minor glitches.

```
bash
sudo systemctl restart saslauthd
sudo systemctl status saslauthd
```

3. If it's still not active after a restart, **review its configuration.** Check the configuration file `/etc/default/saslauthd` to ensure there are no errors or misconfigurations.
4. **Check for multiple installations:** Ensure you don't have multiple installations or versions of `saslauthd` that might be conflicting.

If all else fails, there might be deeper issues with the `saslauthd` installation or configuration on your system, and you'd have to dig deeper or consider purging and reinstalling it.



```
tom@plan6:~$ swaks --to twcookson@gmail.com --from billing@plan6.com --server 74.208.253.216:25 --auth LOGIN --auth-user billing --auth-password
redacted --tls === Trying 74.208.253.216:25... === Connected to 74.208.253.216. <- 220 plan6.com ESMTP Exim 4.94.2 Sun, 03 Sep 2023 23:55:40
+0000 -> EHLO plan6.com <- 250-plan6.com Hello plan6.com [74.208.253.216] <- 250-SIZE 52428800 <- 250-8BITMIME <- 250-PIPELINING <- 250-
PIPE_CONNECT <- 250-CHUNKING <- 250-STARTTLS <- 250-PRDR <- 250 HELP -> STARTTLS <- 220 TLS go ahead === TLS started with cipher
TLSv1.3:TLS_AES_256_GCM_SHA384:256 === TLS no local certificate set === TLS peer DN="/C=UK/O=Exim Developers/CN=plan6.com" -> EHLO
plan6.com <- 250-plan6.com Hello plan6.com [74.208.253.216] <- 250-SIZE 52428800 <- 250-8BITMIME <- 250-PIPELINING <- 250-PIPE_CONNECT
<- 250-CHUNKING <- 250-PRDR <- 250 HELP *** Host did not advertise authentication -> QUIT <- 221 plan6.com closing connection === Connection
closed with remote host. Explain this for me
```



Certainly. Let's break down the interaction step by step:

1. **Connection Initiation:**

```
diff
=== Trying 74.208.253.216:25...
=== Connected to 74.208.253.216.
```

`swaks` is attempting to connect to the server at IP address `74.208.253.216` on port `25`. The connection is successfully established.

2. **Server Greeting:**

```
yaml
<- 220 plan6.com ESMTP Exim 4.94.2 Sun, 03 Sep 2023 23:55:40 +0000
```

The server introduces itself. It's running the Exim version `4.94.2` and is ready to start an SMTP session.

3. **EHLO Command:**

```
diff
-> EHLO plan6.com
```

The `EHLO` command is sent by the client (`swaks`) to the server. This command identifies the sending SMTP server and requests the receiving server to list its extended features.

4. **Server Response to EHLO:** The server responds with a list of capabilities it supports:

- SIZE 52428800: Maximum message size is approximately 50MB.
- 8BITMIME: Supports 8-bit data transmission.
- PIPELINING: Can process multiple commands without waiting for a response to each one.
- CHUNKING: Can accept large emails in smaller chunks.
- STARTTLS: Supports upgrading the connection to a secure TLS connection. ...and so on.

#### 5. STARTTLS Command:

```
diff
-> STARTTLS
```

swaks requests to switch the connection to TLS mode (encrypted). This is to secure the communication.

#### 6. Server Response to STARTTLS:

```
go
<- 220 TLS go ahead
```

The server agrees and indicates that it's ready to start the TLS handshake.

#### 7. TLS Handshake & Encryption:

```
sql
=== TLS started with cipher TLSv1.3:TLS_AES_256_GCM_SHA384:256
=== TLS no local certificate set
=== TLS peer DN="/C=UK/O=Exim Developers/CN=plan6.com"
```

The TLS handshake is successfully negotiated with the mentioned cipher suite. The server presents its certificate to the client for validation.

#### 8. Second EHLO Command:

```
shell
~> EHLO plan6.com
```

After the TLS encryption is set up, swaks sends another EHLO command. This is standard procedure because capabilities might differ after switching to a secure connection.

#### 9. Server's Second Response to EHLO: It's basically the same list of capabilities, but note that STARTTLS isn't offered again since the connection is already secured.

#### 10. Issue Alert:

```
diff
*** Host did not advertise authentication
```

This is where the problem is. Even though you have provided the `--auth LOGIN` option, the server (after switching to TLS) didn't advertise any authentication mechanisms. This means swaks wasn't told by the server how it can authenticate.

#### 11. Session Termination:

```
shell
~> QUIT
```

swaks gracefully terminates the session as it can't proceed without authentication.

**In conclusion:** The initial handshake and TLS setup are working fine. The issue arises when the server doesn't advertise any authentication mechanisms after the TLS setup. You might need to check the Exim configuration related to authentication to resolve this.



You:

```
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: self-signed certificate (_ssl.c:1007)
```



ChatGPT:

The error you're encountering (`ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: self-signed certificate`) means that the Python code you're running is having trouble verifying the SSL certificate of the server it's trying to communicate with. Specifically, it looks like the server is using a self-signed certificate.

A self-signed certificate is a certificate that is not signed by a trusted Certificate Authority (CA). While these certificates provide encryption, they don't provide authentication, meaning you can't trust them to confirm the server's identity. Because of this lack of third-party validation, self-signed certificates are considered insecure for public websites and services.

To resolve or work around this error:

1. **Install the certificate on your machine:** If you trust the source and you know the server should be using a self-signed certificate, you can manually add the certificate to your system's trust store. This tells your system to trust the certificate in future communications.
2. **Use a Certificate from a Trusted CA:** If you have control over the server, it's better to get a certificate from a trusted CA. Let's Encrypt offers free certificates.
3. **Disable Certificate Verification** (Not Recommended for Production!): For temporary development or testing purposes only, you can disable certificate verification in Python:

```
python
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

Remember, this makes your connection insecure because you're bypassing the verification that ensures you're talking to who you think you're talking to. You should never do this in production.

4. **Specify the Certificate Path:** If you know where the certificate is, you can specify the path to use that particular certificate for verification:

```
python
import requests
response = requests.get('https://yourserver.com', verify='/path/to/certfile')
```

Choose the method that best suits your situation. If this is a production environment or involves sensitive data, strongly consider using a certificate from a trusted CA.



You:

give me a mailx usage example



ChatGPT:

Certainly! mailx is a simple command-line email client available on many Unix-like systems.

Here's a basic usage example:

#### 1. Sending an Email:

```
bash
echo "This is the body of the email" | mailx -s "Subject Here" recipient@example.com
```

#### 2. Sending an Email with an Attachment:

```
bash
echo "See attached file" | mailx -s "Subject with Attachment" -a /path/to/attachment.txt recipient@example.com
```

#### 3. Sending an Email Using a Specified SMTP Server:

This usually requires configuring mailx with an SMTP server. For example, if you're using s-nail (a variant of mailx), you might have a configuration in ~/.mailrc like:

```
arduino
set smtp=smtp://smtp.example.com:587
set smtp-auth=login
set smtp-auth-user=your_username
set smtp-auth-password=your_password
set ssl-verify=ignore
```

Then, you'd send mail as:

```
bash
echo "Body of the email" | mailx -s "SMTP Subject" recipient@example.com
```

#### 4. Reading Emails:

Simply invoke `mailx` without arguments:

```
bash
mailx
```

Inside the `mailx` interface, you can navigate through emails, delete them, reply to them, etc. Use the `?` command once inside to see available commands.

Remember, the actual options and capabilities of `mailx` might vary slightly depending on the implementation or version you have installed. It's always a good idea to check the man page (`man mailx`) or the built-in help for specifics.

-----



how view postfix last 40 log

-----



To view the last 40 lines of the Postfix logs, you'll typically want to look into the `/var/log/mail.log` file (on systems that use `rsyslog` to handle logging for Postfix). Here's how you can do it:

```
bash
tail -n 40 /var/log/mail.log
```

This will display the last 40 lines of the log. If your system logs Postfix's mail logs to a different location or file, adjust the file path accordingly.

If your system uses log rotation (which most systems do), older logs might be in files like `/var/log/mail.log.1`, `/var/log/mail.log.2.gz`, etc. If you want to see older logs, you can use tools like `zcat` or `zless` for the gzipped files:

```
bash
zcat /var/log/mail.log.2.gz | tail -n 40
```

Replace `mail.log.2.gz` with the appropriate filename if you're looking into a different log archive.

-----



explain this in postfix log: Sep 4 02:33:54 plan6 postfix/pickup[22547]: ADA4042C58: uid=1000 from=<tom> Sep 4 02:33:54 plan6 postfix/cleanup[22640]: ADA4042C58: message-id=<20230904023354.ADA4042C58@plan6.com> Sep 4 02:33:54 plan6 postfix/qmgr[22548]: ADA4042C58: from=<tom@plan6.com>, size=397, nrcpt=1 (queue active) Sep 4 02:33:54 plan6 postfix/smtp[22642]: connect to plan6.com[74.208.253.216]:587: Connection refused Sep 4 02:33:54 plan6 postfix/smtp[22642]: ADA4042C58: to=<twcookson@gmail.com>, relay=none, delay=0.07, delays=0.03/0.04/0/0, dsn=4.4.1, status=deferred (connect to plan6.com[74.208.253.216]:587: Connection refused)

-----



Sure! Here's a breakdown of the log entries you've provided:

- ```
bash
Sep  4 02:33:54 plan6 postfix/pickup[22547]: ADA4042C58: uid=1000 from=<tom>
```

  - `Sep 4 02:33:54` is the timestamp of the log entry.
  - `plan6` is the hostname of the machine.
  - `postfix/pickup[22547]` is the Postfix pickup service with the process ID 22547.
  - `ADA4042C58` is the internal message ID assigned by Postfix.
  - `uid=1000` means the email was submitted by the user with user ID 1000, which typically corresponds to a non-root user (often the first user created on a system).
  - `from=<tom>` is the envelope sender of the email.
- ```
swift
Sep  4 02:33:54 plan6 postfix/cleanup[22640]: ADA4042C58: message-id=<20230904023354.ADA4042C58@plan6.com>
```

- The postfix/cleanup service with process ID 22640 has processed the message with internal ID ADA4042C58.
- The globally unique message-id for this email is <20230904023354.ADA4042C58@plan6.com>.

### 3. arduino

```
Sep  4 02:33:54 plan6 postfix/qmgr[22548]: ADA4042C58: from=<tom@plan6.com>, size=397, nrcpt=1 (queue active)
```

- The postfix/qmgr service (queue manager) with process ID 22548 is handling the email.
- The envelope sender address is given as tom@plan6.com.
- The message size is 397 bytes.
- There's 1 recipient (nrcpt=1).

### 4. less

```
Sep  4 02:33:54 plan6 postfix/smtp[22642]: connect to plan6.com[74.208.253.216]:587: Connection refused
```

- The postfix/smtp service (with process ID 22642) attempted to connect to the SMTP server at plan6.com on IP 74.208.253.216 using port 587.
- The connection was refused, indicating that nothing is listening on that port, or a firewall is blocking the connection.

### 5. sql

```
Sep  4 02:33:54 plan6 postfix/smtp[22642]: ADA4042C58: to=<twcookson@gmail.com>, relay=none, delay=0.07, dela
```

- This line elaborates on the connection issue.
- The message was intended for twcookson@gmail.com.
- relay=none indicates that no SMTP relay was used since the connection failed.
- The delay and delays fields provide timing details.
- dsn=4.4.1 is a Delivery Status Notification code, where 4.4.1 typically means a transient failure connecting to the host.
- The message has a status=deferred, meaning Postfix will attempt to send the message again later.
- The reason provided for the deferment is the earlier connection refusal.

In essence, your Postfix server tried to send an email from tom@plan6.com to twcookson@gmail.com, but it couldn't connect to the SMTP server on plan6.com at port 587. The message will remain in the Postfix queue, and the system will try to send it again later.



You:

How do i check if its listening on 587 in general?



ChatGPT:

To check if a service is listening on port 587 (or any other port) on your system, you can use the netstat or ss commands, combined with grep to filter the results. Here's how to do it:

#### 1. Using netstat:

If you don't have netstat installed, it comes with the net-tools package. You can install it on a Debian/Ubuntu system with:

```
sudo apt install net-tools
```

Once installed, run:

```
perl
netstat -tuln | grep 587
```

#### 2. Using ss:

ss is a utility to investigate sockets and is installed by default on most modern Linux distributions.

```
perl
ss -tuln | grep 587
```

Either of the above commands will display services listening on port 587. If nothing is returned from the command, then nothing is currently listening on that port.

#### 3. Using lsof:

The lsof command (list open files) can also be used to check for listening ports:

```
css
sudo lsof -i :587
```

If a service is listening on port 587, one of these commands will provide information about it. If the commands return no output, then no service is currently listening on that port.



You:

Nothing is listening on 587, how do I fix that? I do see activity on 25



ChatGPT:

If you see activity on port 25, it means Postfix is set up for SMTP on the default port. However, port 587 is generally used for client-to-server communication (submission), especially when authentication is involved. To enable this on Postfix:

#### 1. Edit the Postfix master.cf file:

```
bash
sudo nano /etc/postfix/master.cf
```

#### 2. Uncomment or add the submission section:

You might see a section like this, but it could be commented out (lines starting with #).

```
shell
#submission inet n      -       y       -       -       smtpd
#  -o syslog_name=postfix/submission
#  -o smtpd_tls_security_level=encrypt
#  -o smtpd_sasl_auth_enable=yes
#  -o smtpd_relay_restrictions=permit_sasl_authenticated,reject
#  -o milter_macro_daemon_name=ORIGINATING
```

Uncomment these lines by removing the # at the beginning of each line. It should look like:

```
bash
submission inet n      -       y       -       -       smtpd
  -o syslog_name=postfix/submission
  -o smtpd_tls_security_level=encrypt
  -o smtpd_sasl_auth_enable=yes
  -o smtpd_relay_restrictions=permit_sasl_authenticated,reject
  -o milter_macro_daemon_name=ORIGINATING
```

#### 3. Save and close the file:

If you're using nano, press CTRL + X to close, Y to confirm changes, and Enter to save.

#### 4. Reload or restart Postfix:

```
bash
sudo systemctl reload postfix
```

#### 5. Check if Postfix is now listening on port 587:

```
bash
netstat -tuln | grep 587
```

or

```
bash
ss -tuln | grep 587
```

If configured correctly, Postfix should now be listening on port 587 for mail submissions.





You:

```
Sep 4 02:40:38 plan6 postfix/postfix-script[22722]: refreshing the Postfix mail system Sep 4 02:40:38 plan6 postfix/master[22545]: reload -- version 3.5.18, configuration /etc/postfix
Sep 4 02:40:38 plan6 postfix/qmgr[22727]: BECDE42C4C: from=<tom@plan6.com>, size=397, nrcpt=1 (queue active) Sep 4 02:40:38 plan6 postfix/qmgr[22727]: ADA4042C5B: from=<tom@plan6.com>, size=397, nrcpt=1 (queue active) Sep 4 02:40:39 plan6 postfix/smtp[22730]: connect to plan6.com[74.208.253.216]:587: Connection refused Sep 4 02:40:39 plan6 postfix/smtp[22732]: connect to plan6.com[74.208.253.216]:587: Connection refused Sep 4 02:40:39 plan6 postfix/smtp[22730]: BECDE42C4C: to=<twcookson@gmail.com>, relay=none, delay=888, delays=888/0.09/0.04/0, dsn=4.4.1, status=deferred (connect to plan6.com[74.208.253.216]:587: Connection refused) Sep 4 02:40:39 plan6 postfix/smtp[22732]: ADA4042C5B: to=<twcookson@gmail.com>, relay=none, delay=404, delays=404/0.1/0.02/0, dsn=4.4.1, status=deferred (connect to plan6.com[74.208.253.216]:587: Connection refused) Sep 4 02:40:55 plan6 postfix/pickup[22728]: 5638542C5B: uid=1000 from=<tom> Sep 4 02:40:55 plan6 postfix/cleanup[22741]: 5638542C5B: message-id=<20230904024055.5638542C5B@plan6.com> Sep 4 02:40:55 plan6 postfix/qmgr[22727]: 5638542C5B: from=<tom@plan6.com>, size=397, nrcpt=1 (queue active) Sep 4 02:40:55 plan6 postfix/smtp[22730]: connect to plan6.com[74.208.253.216]:587: Connection refused Sep 4 02:40:55 plan6 postfix/smtp[22730]: 5638542C5B: to=<twcookson@gmail.com>, relay=none, delay=0.05, delays=0.05/0/0/0, dsn=4.4.1, status=deferred (connect to plan6.com[74.208.253.216]:587: Connection refused) tom@plan6:~$ netstat -tuln | grep 587 tcp 0 0 127.0.0.1:587 0.0.0.0:* LISTEN tcp6 0 0 :::587 :::* LISTEN tom@plan6:~$
```



ChatGPT:

Based on the logs and the output of `netstat`, here's the interpretation:

#### 1. Logs:

- Postfix tried to send out emails but the connection to `plan6.com` on port 587 is refused.
- The system has attempted to send the message multiple times (as evident by the `status=deferred` log entries). The `dsn=4.4.1` refers to the specific error message: "Connection refused".

#### 2. netstat output:

- Postfix is indeed listening on port 587, but it's bound to the localhost addresses (`127.0.0.1` for IPv4 and `:::1` for IPv6). This means it's only accepting connections from the local machine on that port.

The issue here is that Postfix, configured as the client, is trying to relay emails through port 587, which is generally used for secure client-to-server communication (submission), and expects to find a mail submission server there. However, it looks like your Postfix setup is also the target of this connection.

### Possible Solutions:

#### 1. Use a different relay:

If you intended to use another server as a relay, you need to configure Postfix to use the correct relay address and port in `main.cf`:

```
css
relayhost = [some.other.server]:587
```

#### 2. Bind to All Addresses:

If you intended to have Postfix accept external connections on port 587, you'll need to adjust the `master.cf` file to make it listen on all network interfaces. Update the line:

```
submission inet n      -      y      -      -      smtpd
```

To:

```
submission inet n      -      y      -      -      smtpd -o smtpd_bind_address=0.0.0.0
```

Then reload Postfix:

```
swift
sudo systemctl reload postfix
```

#### 3. Adjust Firewall:

Ensure that any firewalls running on the server (like `ufw` or `iptables`) are not blocking incoming connections on port 587.

#### 4. Review Relay Restrictions:

If you're trying to send email from `tom@plan6.com` to an external email address through your own mail server, ensure that `smtpd_relay_restrictions` in the `main.cf` configuration file is allowing for authenticated users or the local network to relay mail.

#### 5. Check DNS:

Ensure that the hostname (plan6.com) resolves correctly. If Postfix is trying to send email via a relay server using the domain name, but DNS resolution points back to the same server, there might be a misconfiguration.

To further troubleshoot, you might consider checking main.cf for relay settings and verifying the intended behavior.



tom@plan6:~\$ netstat -tuln | grep 587 tcp 0 0 127.0.0.1:587 0.0.0.0:\* LISTEN tcp6 0 0 :::587 :::\* LISTEN tom@plan6:~\$ I want it listening from the outside world as well]



If you want Postfix to listen on port 587 from outside as well, you need to adjust its configuration. Currently, Postfix is only listening on the localhost (127.0.0.1 for IPv4 and ::1 for IPv6), which means it only accepts connections from the local machine.

Here's how you can change that:

1. Open the Postfix master configuration file:

```
bash
sudo nano /etc/postfix/master.cf
```

2. Find the line for the submission service. It should look like this:

```
submission inet n      -      y      -      -      smtpd
```

3. By default, the submission line doesn't specify a binding address, which causes Postfix to listen on localhost. Modify this line by adding the -o smtpd\_bind\_address=0.0.0.0 option, which will bind it to all available network interfaces. The modified line should look like this:

```
submission inet n      -      y      -      -      smtpd -o smtpd_bind_address=0.0.0.0
```

This will make Postfix listen on all IPv4 addresses on the machine.

If you also want it to listen on all IPv6 addresses, add -o smtpd\_bind\_address\_ipv6=:::

```
submission inet n      -      y      -      -      smtpd -o smtpd_bind_address=0.0.0.0 -o smtpd_bind_add
```

4. Save the file and close the editor (with nano, it's CTRL + X to close, press Y to confirm changes, and then press Enter).
5. Reload the Postfix configuration:

```
bash
sudo systemctl reload postfix
```

6. Check again using netstat:

```
bash
netstat -tuln | grep 587
```

You should now see the service listening on 0.0.0.0:587 and :::587.

7. Lastly, make sure your firewall (if active) allows incoming connections on port 587. If you're using ufw, you can allow this with:

```
bash
sudo ufw allow 587/tcp
```

Now, Postfix should be listening on port 587 from both inside and outside.



tom@plan6:~\$ netstat -tuln | grep 587 tcp 0 0 127.0.0.1:587 0.0.0.0:\* LISTEN tcp6 0 0 :::587 :::\* LISTEN tom@plan6:~\$



It seems that even after making the configuration changes, Postfix is still only listening on the localhost (127.0.0.1 for IPv4 and ::1 for IPv6).

Let's try another approach:

1. Open the Postfix main configuration file:

```
bash
sudo nano /etc/postfix/main.cf
```

2. Find the `inet_interfaces` parameter. It will probably be set as:

```
makefile
inet_interfaces = localhost
```

3. Change the `inet_interfaces` parameter to:

```
css
inet_interfaces = all
```

This will make Postfix listen on all network interfaces. If you only have a specific IP you want to bind to, you can also replace `all` with that IP address.

4. Save the file and close the editor.
5. Reload the Postfix configuration:

```
bash
sudo systemctl reload postfix
```

6. Check again using `netstat`:

```
bash
netstat -tuln | grep 587
```

You should now see port 587 listening on 0.0.0.0 (all IPv4 addresses) and :: (all IPv6 addresses).

If this still doesn't work, it's possible there are other parameters or configurations overriding these settings. In that case, you might need to dive deeper into the configuration files or consult any Postfix customization specific to your system.



You:

```
# TLS parameters smtpd_tls_cert_file=/etc/letsencrypt/live/plan6.com/fullchain.pem smtpd_tls_key_file=/etc/letsencrypt/live/plan6.com/privkey.pem
smtpd_tls_security_level=may smtpd_tls_CApath=/etc/ssl/certs smtpd_tls_security_level=may smtpd_tls_session_cache_database =
btree:${data_directory}/smtpd_scache smtpd_relay_restrictions = permit_mynetworks permit_sasl_authenticated defer_unauth> myhostname = plan6.com
alias_maps = hash:/etc/aliases alias_database = hash:/etc/aliases myorigin = /etc/mailname mydestination = $myhostname, plan6.com, localhost.com, ,
localhost relayhost = mynetworks = 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128 mailbox_size_limit = 0 recipient_delimiter = + inet_interfaces = all
inet_protocols = all # enable SASL authentication smtp_sasl_auth_enable = yes # disallow methods that allow anonymous authentication.
smtp_sasl_security_options = noanonymous # where to find sasl_passwd smtp_sasl_password_maps = hash:/etc/postfix/sasl_passwd # Enable
STARTTLS encryption smtp_use_tls = yes # where to find CA certificates smtp_tls_CAfile = /etc/ssl/certs/ca-certificates.crt
```



Your Postfix configuration seems fine for the most part, especially in terms of listening on all interfaces (`inet_interfaces = all`) and for both IPv4 and IPv6 protocols (`inet_protocols = all`).

However, a few things need to be checked:

1. **Permissions on TLS files:** Make sure that the Let's Encrypt certificates you've specified are readable by the Postfix user. Typically, they should be readable by root and a group that Postfix is a part of. You can check permissions with:

```
bash
```

```
ls -l /etc/letsencrypt/live/plan6.com/
```

2. **Relay Restrictions:** The `smtpd_relay_restrictions` entry seems to be cut off (`defer_unauth>`). Make sure you have the complete list of relay restrictions.
3. **SASL Configuration:** Since you're using SASL authentication, ensure `saslauthd` is running and configured properly. You can check its status with:

```
bash
sudo systemctl status saslauthd
```

4. **Firewall:** Ensure there's no firewall rule blocking port 587. If you're using `ufw`, you can list the rules with `sudo ufw status`.
5. **Restart Postfix:** Given that configuration changes may not always be picked up with a reload, try restarting Postfix completely:

```
bash
sudo systemctl restart postfix
```

6. **Check Listening Ports Again:** After restarting Postfix, check if it's listening on port 587 for all interfaces:

```
bash
netstat -tln | grep 587
```

If none of these suggestions resolve the issue, it might be helpful to review the Postfix logs for any error messages or clues as to why it's not listening on the desired interfaces. Typically, these logs are in `/var/log/mail.log` or `/var/log/maillog`. Remember to also check for any other configuration files being imported or included in `main.cf` that might override the specified settings.



You:

Also: `#delay_warning_time = 4h` `readme_directory = no` # See [http://www.postfix.org/COMPATIBILITY\\_README.html](http://www.postfix.org/COMPATIBILITY_README.html) -- default to 2 on # fresh installs.  
`compatibility_level = 2` # TLS parameters `smtpd_tls_cert_file=/etc/letsencrypt/live/plan6.com/fullchain.pem`  
`smtpd_tls_key_file=/etc/letsencrypt/live/plan6.com/privkey.pem` `smtpd_tls_security_level=may` `smtp_tls_CAspath=/etc/ssl/certs`  
`smtp_tls_security_level=may` `smtp_tls_session_cache_database = btree:${data_directory}/smtp_scache` `smtpd_relay_restrictions = permit_mynetworks`  
`permit_sasl_authenticated defer_unauth>` `myhostname = plan6.com` `alias_maps = hash:/etc/aliases` `alias_database = hash:/etc/aliases` `myorigin =`  
`/etc/mailname` `mydestination = $myhostname, plan6.com, localhost.com, , localhost` `relayhost = mynetworks = 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128`  
`mailbox_size_limit = 0` `recipient_delimiter = +` `inet_interfaces = all` `inet_protocols = all`



ChatGPT:

Thanks for the additional configuration details. Let's look specifically at the `smtpd` configurations which relate to the SMTP server daemon used for incoming mail, including submission on port 587.

From your current configuration, I do see that:

1. You have `inet_interfaces = all` which means Postfix will bind to all available network interfaces.
2. You're using Let's Encrypt for your TLS configuration, which is a great choice. Ensure that the certificates are valid and not expired, and as mentioned earlier, that their permissions are set correctly.
3. `smtpd_relay_restrictions` seems to still be cut off (`defer_unauth>`). This might be an issue.

Let's start by adding the submission service for port 587 with encryption:

1. **Configure submission in master.cf:** Open the `/etc/postfix/master.cf` file, and ensure you have a section for submission. This section is often commented out by default. It should look something like:

```
bash
submission inet n      -      y      -      -      smtpd
-o syslog_name=postfix/submission
-o smtpd_tls_security_level=encrypt
-o smtpd_sasl_auth_enable=yes
-o smtpd_reject_unlisted_recipient=no
...
```

If this section is commented out (lines starting with `#`), uncomment it.

2. **Ensure Proper SASL Configuration:** If you're enabling SASL for the submission service, ensure you have `saslauthd` or the equivalent SASL service running and configured correctly. The `smtpd_sasl_auth_enable=yes` option requires this.
3. **Restart Postfix:**

```
bash
sudo systemctl restart postfix
```

#### 4. Check the Listening Ports Again:

```
bash
netstat -tuln | grep 587
```

This should hopefully have Postfix listening on port 587. If not, it would be really helpful to take a look at the logs as they will give a clearer indication of what might be going wrong. Check `/var/log/mail.log` or `/var/log/maillog` after attempting to restart Postfix.



```
tom@plan6:~$ sudo ls -l /etc/letsencrypt/live/plan6.com/ total 4 lrwxrwxrwx 1 root root 33 Aug 10 22:29 cert.pem -> ../../archive/plan6.com/cert2.pem
lrwxrwxrwx 1 root root 34 Aug 10 22:29 chain.pem -> ../../archive/plan6.com/chain2.pem lrwxrwxrwx 1 root root 38 Aug 10 22:29 fullchain.pem ->
../../archive/plan6.com/fullchain2.pem lrwxrwxrwx 1 root root 36 Aug 10 22:29 privkey.pem -> ../../archive/plan6.com/privkey2.pem -rw-r--r-- 1 root root 692
Jun 11 19:46 README tom@plan6:~$
```



It looks like the Let's Encrypt certificates and keys for `plan6.com` are properly symlinked to the actual files in the archive directory. This is the typical structure when using the Certbot tool from Let's Encrypt.

Here are some things to note and check:

##### 1. Permissions:

- The symlinks and the target files they point to should be readable by the `postfix` user or the group it belongs to. This is to ensure that the Postfix process can actually read the certificates and keys when it starts.
- You can check the permissions on the actual files in the archive directory by following the symlink. For instance, `ls -l /etc/letsencrypt/archive/plan6.com/` should give you a list.

##### 2. Certificate Validity:

- Ensure the certificates haven't expired. You can use the `openssl` command for this:

```
bash
openssl x509 -in /etc/letsencrypt/live/plan6.com/cert.pem -noout -dates
```

This will give you the start and end validity dates for the certificate.

##### 3. Ownership and Group:

- Check the ownership and group of the Postfix process and ensure that it has access to read the certificate and key files. Typically, the Postfix process runs as the `postfix` user.

##### 4. Logs:

- If Postfix has issues with the certificates, it would usually log these errors. Check your `/var/log/mail.log` or similar for any TLS or certificate related errors upon Postfix startup or when trying to establish a TLS connection.

##### 5. Testing:

- Test the TLS configuration with `openssl`:

```
bash
openssl s_client -connect localhost:587 -starttls smtp
```

This command should return the certificate chain and some details about the encryption. Look out for any errors or issues reported.

If everything seems okay with the certificates and keys, and Postfix is still not listening on port 587 (as per your previous issue), then the issue might be elsewhere, and we should take a look at other configurations or potential errors in the logs.



```
tom@plan6:~$ sudo openssl x509 -in /etc/letsencrypt/live/plan6.com/cert.pem -noout -dates notBefore=Aug 10 21:29:32 2023 GMT notAfter=Nov 8 21:29:31 2023 GMT tom@plan6:~$
```



The certificate is valid from August 10th, 2023 to November 8th, 2023. This means that your certificate is currently valid.

To further address the issue with port 587:

#### 1. Confirm Port Configuration:

- In your Postfix main configuration (/etc/postfix/main.cf), you should have a master.cf file that defines services and ports. Make sure that submission (SMTP over TLS, typically on port 587) is uncommented and set up correctly in /etc/postfix/master.cf.

```
bash
sudo nano /etc/postfix/master.cf
```

- You should see something like:

```
plaintext
submission inet n      -      y      -      -      smtpd
-o syslog_name=postfix/submission
-o smtpd_tls_security_level=encrypt
-o smtpd_sasl_auth_enable=yes
-o smtpd_client_restrictions=permit_sasl_authenticated,reject
...
```

- If it's commented out (lines starting with #), uncomment it. If it's missing, add it.

#### 2. Firewall Settings:

- Make sure there's no firewall (like UFW or iptables) blocking external access to port 587.

#### 3. Restart Postfix:

- After making changes, always remember to restart or reload Postfix to apply them:

```
bash
sudo systemctl restart postfix
```

#### 4. Check for Errors:

- Always refer to the logs after making changes or if something isn't working as expected. Postfix will generally log helpful error messages or warnings.

```
bash
sudo tail -f /var/log/mail.log
```

- This command will "follow" the log, printing new entries as they appear. It's useful for spotting issues as they occur.

#### 5. Test Port 587 Again:

- Once you've confirmed the configuration and restarted Postfix, try testing port 587 again:

```
bash
netstat -tuln | grep 587
```

If after following these steps, you're still having issues, please let me know. We can further troubleshoot the issue.



You:

```
tom@plan6:~$ sudo systemctl restart postfix tom@plan6:~$ netstat -tuln | grep 587 tcp 0 0 0.0.0.0:587 0.0.0.0:* LISTEN tcp6 0 0 :::587 :::* LISTEN
tom@plan6:~$
```



Great! It looks like Postfix is now listening on port 587 for both IPv4 (0.0.0.0:587) and IPv6 (:::587), meaning it's open to connections from the outside world on that port.

You should now test to see if you can send mail over port 587 using a mail client or another testing tool.

If you experience any issues while sending mail or if you see any errors or warnings in the `/var/log/mail.log` log file, please share them so we can further assist you. Otherwise, it seems that your Postfix is now correctly configured to listen on port 587.



Sep 4 02:57:03 plan6 postfix/submission/smtpd[23462]: connect from ool-ad02faee.dyn.optonline.net[173.2.250.238] Sep 4 02:57:04 plan6 postfix/submission/smtpd[23462]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL CRAM-MD5 authentication failed: authentication failure Sep 4 02:57:04 plan6 postfix/submission/smtpd[23462]: warning: SASL authentication failure: Password verification failed Sep 4 02:57:04 plan6 postfix/submission/smtpd[23462]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL PLAIN authentication failed: authentication failure Sep 4 02:57:04 plan6 postfix/submission/smtpd[23462]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL LOGIN authentication failed: authentication failure Sep 4 02:57:04 plan6 postfix/submission/smtpd[23462]: disconnect from ool-ad02faee.dyn.optonline.net[173.2.250.238] ehlo=2 starttls=1 auth=0/3 quit=1 commands=4/7 tom@plan6:~\$



The logs indicate there were failed authentication attempts when trying to send mail via the submission port (587). Specifically, the client attempted to authenticate using the CRAM-MD5, PLAIN, and LOGIN SASL mechanisms, but all of them failed.

Here are the steps you can take to troubleshoot this:

1. **Check SASL Configuration:** Ensure that you have the correct SASL configuration in place. Since you're using Postfix, the SASL authentication is usually handled by either `cyrus-sasl` or `dovecot`. The configuration will depend on which one you're using.
2. **Correct Credentials:** If you're testing with a mail client or another tool, double-check that you are using the correct username and password.
3. **SASL Mechanism Support:** Some mail servers or clients might not support all SASL mechanisms. You might need to configure your mail server or client to use a specific SASL mechanism. For instance, if you know that CRAM-MD5 is causing issues, you might want to disable it or use another mechanism that both the client and server support.
4. **Logs:** Check more detailed logs on your system related to SASL authentication. Depending on your setup, the logs might be in `/var/log/mail.log`, `/var/log/auth.log`, `/var/log/dovecot.log` or others.
5. **Password Database:** Ensure that the username and password being used to authenticate are correctly set up in your SASL password database. The location of this will depend on your setup, but often it's specified in the `sasl_passwd` file in the Postfix configuration.
6. **Encryption:** When using PLAIN authentication, ensure that the connection is encrypted using STARTTLS (or another encryption method), as transmitting plain text passwords over an unencrypted connection is a security risk. Some servers are configured to reject PLAIN authentication if the connection isn't encrypted.
7. **Check Dovecot (if using it for SASL):** If you're using Dovecot for SASL authentication, ensure that the Dovecot service is running and that the Postfix-to-Dovecot SASL integration is correctly configured.

After making changes, remember to reload or restart your Postfix to apply the changes.

Lastly, it's also a good idea to be cautious and monitor for repeated authentication failures, as they can sometimes indicate someone is trying to guess passwords (a "dictionary" or "brute force" attack).



Write a python script that will very likely handle test the smtp on the other server well



Sure! Below is a basic Python script using the `smtpLib` library to test SMTP authentication and send an email:

```
python
import smtplib
from email.message import EmailMessage
```

```
def send_test_email(smtp_server, port, sender_email, sender_password, recipient_email):
    # Create an email message
    msg = EmailMessage()
    msg.set_content("This is a test email sent from a Python script.")
    msg["Subject"] = "Test Email from Python Script"
    msg["From"] = sender_email
    msg["To"] = recipient_email

    try:
        # Connect to the SMTP server
        with smtplib.SMTP(smtp_server, port) as server:
            # Upgrade the connection to encrypted SSL/TLS
            server.starttls()

            # Login to the SMTP server
            server.login(sender_email, sender_password)

            # Send the email
            server.send_message(msg)

        print(f"Email sent successfully to {recipient_email}!")
    except Exception as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    smtp_server = "plan6.com"
    port = 587
    sender_email = "tom@plan6.com" # Replace with your email
    sender_password = "your_password" # Replace with your password
    recipient_email = "receiver@example.com" # Replace with the recipient's email

    send_test_email(smtp_server, port, sender_email, sender_password, recipient_email)
```

#### Instructions:

1. Replace your\_password with the password for the email tom@plan6.com.
2. Replace receiver@example.com with the email you want to send the test email to.
3. Run the script.

This script will attempt to connect to the SMTP server, authenticate using the provided credentials, and send a test email. If there are any issues during this process, the error will be printed to the console.

*Important:* Make sure you store your password securely and don't commit the password in any public code repositories. Always avoid hardcoding passwords directly into scripts; this script is meant for demonstration and testing purposes only.



You:

in the table of users, I have my user name as billing and my password as pass. Should my username be billing@plan6.com with the password pass?

-----



ChatGPT:

When authenticating with an SMTP server, the expected format of the username often depends on the server's configuration and the email system in place.

#### 1. Username Only (without domain):

- Some email servers are set up to recognize users based solely on the username, especially when they handle mail for only one domain. In this case, you'd use billing as the username.

#### 2. Full Email Address:

- Many email servers, especially those that handle mail for multiple domains, expect the full email address as the username (e.g., billing@plan6.com). This helps differentiate users across different domains.

Given that you're working with Postfix and based on the information you've shared, I would recommend trying both methods:

1. Authenticate with just the username billing and password pass.
2. If that doesn't work, try the full email address billing@plan6.com with the password pass.

In the Python script I provided earlier, you can easily adjust the sender\_email variable between the two formats to see which one works for your server configuration.



In general, it's also a good practice to avoid using overly simple passwords like "pass", especially for email servers, as they can be easily guessed. Consider using a stronger password and, if possible, enable additional security measures like two-factor authentication.



Adjust the python code for the non email address user name format



Certainly! Here's the Python code from earlier adjusted for a non-email address username format:

```
python
import smtplib
from email.mime.text import MIMEText

# SMTP Server Configuration
SMTP_SERVER = 'plan6.com'
SMTP_PORT = 587
USERNAME = 'billing' # Adjusted for non-email format
PASSWORD = 'pass'
SENDER_EMAIL = 'billing@plan6.com'
RECIPIENT_EMAIL = 'your_email@example.com' # Change this to your recipient email
SUBJECT = 'Test Email'
MESSAGE = 'This is a test email sent from a Python script.'

# Create MIMEText object and fill in headers & payload
msg = MIMEText(MESSAGE)
msg['Subject'] = SUBJECT
msg['From'] = SENDER_EMAIL
msg['To'] = RECIPIENT_EMAIL

# Send the email
try:
    with smtplib.SMTP(SMTP_SERVER, SMTP_PORT) as smtp:
        smtp.starttls()
        smtp.login(USERNAME, PASSWORD)
        smtp.sendmail(SENDER_EMAIL, RECIPIENT_EMAIL, msg.as_string())
    print("Email sent successfully!")
except Exception as e:
    print(f"Error sending email: {e}")
```

Please ensure you replace 'your\_email@example.com' with the actual recipient's email address where you want to receive the test email.

Run the script to send an email using the adjusted username format. If it fails, the provided error message might offer more insight into what's going wrong.



```
Sep 4 03:20:08 plan6 postfix/submission/smtpd[23581]: connect from ool-ad02faee.dyn.optonline.net[173.2.250.238] Sep 4 03:20:08 plan6
postfix/submission/smtpd[23581]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL CRAM-MD5 authentication failed: authentication failure
Sep 4 03:20:09 plan6 postfix/submission/smtpd[23581]: warning: SASL authentication failure: Password verification failed Sep 4 03:20:09 plan6
postfix/submission/smtpd[23581]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL PLAIN authentication failed: authentication failure Sep 4
03:20:09 plan6 postfix/submission/smtpd[23581]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL LOGIN authentication failed:
authentication failure Sep 4 03:20:09 plan6 postfix/submission/smtpd[23581]: disconnect from ool-ad02faee.dyn.optonline.net[173.2.250.238] ehlo=2
starttls=1 auth=0/3 quit=1 commands=4/7
```



The logs are showing repeated SASL authentication failures, regardless of the mechanism used (CRAM-MD5, PLAIN, LOGIN). This clearly suggests a problem with your authentication configuration or credentials.

Let's break this down step-by-step:

### 1. SASL Authentication Mechanisms:

- CRAM-MD5: Requires the server to provide a challenge, and clients send a hash that combines the password and challenge.
- PLAIN: Sends the password in plaintext (usually over an encrypted connection).
- LOGIN: Similar to PLAIN but has a slightly different flow.

The logs show that the client is trying different methods to authenticate, but all of them fail.

### 2. Common Reasons for SASL Authentication Failures:

- **Incorrect credentials:** Ensure that the username and password you're using in the Python script are correct. Check for any typos.
- **SASL not properly configured on the server:** The SASL configuration might not be correctly set up on your Postfix server.
- **Password databases not updated or mismatched:** If you recently changed or set up the SASL authentication, it might be worth rebuilding or rechecking the databases related to SASL.
- **Mismatched mechanisms:** Ensure the client (your Python script) and server both support the SASL mechanism being used.

### 3. Troubleshooting Steps:

- Ensure the `sasl_passwd` file (or equivalent if you're not using `sasl_passwd`) contains the correct credentials and has been postmapped (using `postmap` command).
- Ensure the Postfix SASL configuration (`smtpd_sasl_path`, `smtpd_sasl_type`, etc.) is correct.
- If using `saslauthd` or another mechanism for authentication, ensure the service is running and correctly configured.
- Check for any restrictions or filters that might be blocking authentication attempts, such as firewalls or `smtpd_recipient_restrictions` in the Postfix config.
- Use tools like `testsaslauthd` (if you're using `saslauthd`) to test authentication locally on the server.

Finally, if the issue persists, you might need to delve deeper into your mail server's configuration and logs. Adjusting log verbosity could provide more insight into the failure reason.



You:

how do I add a new username and password?



ChatGPT:

To add a new username and password for SASL authentication in Postfix, follow these steps:

#### 1. Adding User to `sasl_passwd` file:

First, you need to choose the location for your `sasl_passwd` file. A common location is `/etc/postfix/sasl_passwd`. If this file does not already exist, you'll create it. If it does, you'll add to it.

Add the credentials in this format:

```
markdown
[your.mail.server]:port username:password
```

For example, if you want to add the user `billing` with password `pass`, and your mail server is `smtp.plan6.com` on port 587, then:

```
bash
echo "[smtp.plan6.com]:587 billing:pass" >> /etc/postfix/sasl_passwd
```

**Note:** Replace the mail server domain and port with your actual SMTP relay's domain and port.

#### 2. Create a db file for Postfix:

Postfix does not read the `sasl_passwd` file directly. Instead, it reads from a database file generated from the `sasl_passwd` file.

To generate this db file:

```
bash
postmap /etc/postfix/sasl_passwd
```

This will create a file named `/etc/postfix/sasl_passwd.db`.

#### 3. Secure the files:

For security reasons, ensure that the `.db` file and the original `sasl_passwd` file are readable only by the root user:

```
bash
```

```
sudo chown root:root /etc/postfix/sasl_passwd /etc/postfix/sasl_passwd.db
sudo chmod 0600 /etc/postfix/sasl_passwd /etc/postfix/sasl_passwd.db
```

#### 4. Reload or Restart Postfix:

After adding new users or making changes, ensure you reload or restart the Postfix service:

```
bash
sudo systemctl reload postfix
```

Or, if you prefer:

```
bash
sudo service postfix reload
```

### 5. Test the new credentials:

You can then test the new credentials using the provided Python script, or any SMTP client to ensure that the authentication is working correctly.

Remember to backup important configuration files before making changes, and always be cautious when working with authentication credentials to keep them secure.



```
I think i got closer to sending an email than ever before.... what went right, and what went wrong here: (pback) tom@laptoplin:~/ParKEzAl/backend$ swaks
--to=twcookson@gmail.com --from=billing@plan6.com --server plan6.com:587 --auth --tls Username: billing Password: raccoon === Trying plan6.com:587...
=== Connected to plan6.com. <- 220 plan6.com ESMTP Postfix (Debian/GNU) -> EHLO laptoplin <- 250-plan6.com <- 250-PIPELINING <- 250-SIZE
10240000 <- 250-VRFY <- 250-ETRN <- 250-STARTTLS <- 250-ENHANCEDSTATUSCODES <- 250-8BITMIME <- 250-DSN <- 250-SMTPTF8 <- 250
CHUNKING -> STARTTLS <- 220 2.0.0 Ready to start TLS === TLS started with cipher TLSv1.3:TLS_AES_256_GCM_SHA384:256 === TLS no local
certificate set === TLS peer DN="/CN=plan6.com" -> EHLO laptoplin <- 250-plan6.com <- 250-PIPELINING <- 250-SIZE 10240000 <- 250-VRFY <-
250-ETRN <- 250-AUTH DIGEST-MD5 CRAM-MD5 NTLM LOGIN PLAIN <- 250-ENHANCEDSTATUSCODES <- 250-8BITMIME <- 250-DSN <- 250-
SMTPTF8 <- 250 CHUNKING ==> AUTH CRAM-MD5 <- 334 PDQyMjA0ZmY2MDYuMTYwNmZlYVAcGxhbJyUy29tPg== ->
YmlsbGlzZyZyUWnwEwODZhNWrmNDNIoGM1NzJiM2Y4ZWZWM2MTIjOQ== <-* 535 5.7.8 Error: authentication failed: authentication failure -> AUTH
DIGEST-MD5 <- 334
bm9uY2U9ljZyUDfAr2FtSjdmeTINajA5VjJ5TU1Qbk1WaXdJRStKREzaMtJ6SHJ1NFk9lixYZWFsbT0icGxhbJyUy29tlxxb3A9ImF1dGgiLGNoyXJzZXQ9dX
==>
Y2hhcnNldD11dGYtOCxbJm9uY2U9ljhlODI4MGEzYzdlyZrjNjMzNmZmZjZDU2ZGJyZAxlixaWdlc3QtddXJpPSJzbXRwL3BsYW42LmNvbSlSbmM9MDAw
<-* 535 5.7.8 Error: authentication failed: authentication failure -> AUTH LOGIN <- 334 VXNlcm5lbWU6 ==> YmlsbGlzZw== <- 334 UGFzc3dvcmQ6 ->
cmFYj29vbvg== <-* 535 5.7.8 Error: authentication failed: authentication failure -> AUTH PLAIN AGJpbGxpbnmcAcMFjY29vbvg== <-* 535 5.7.8 Error:
authentication failed: authentication failure *** No authentication type succeeded ==> QUIT <- 221 2.0.0 Bye
```



From the logs you've provided, there are several things that went right and a critical piece that went wrong:

### What Went Right:

1. **Connection:** You were able to successfully connect to the SMTP server at `plan6.com` on port 587. The server responded with a 220 greeting, indicating it's ready to start the SMTP dialogue.
2. **TLS/SSL Handshake:** You successfully initiated a STARTTLS command and successfully established a TLS connection, as indicated by the line `=== TLS started with cipher TLSv1.3:TLS_AES_256_GCM_SHA384:256`. This means your server has TLS set up correctly, which encrypts your SMTP connection.
3. **SMTP Commands:** After establishing a secure TLS connection, your client successfully issued another EHLO command, and the server responded with a list of capabilities. This indicates that the TLS wrapper is working fine.

### What Went Wrong:

1. **Authentication:** The main issue here is that every authentication mechanism attempted (CRAM-MD5, DIGEST-MD5, LOGIN, PLAIN) failed with a 535 5.7.8 Error: authentication failed: authentication failure error. This means that the provided username and/or password was incorrect or not recognized, or there could be a misconfiguration in the way Postfix is handling SASL authentication.

### What to do next:

1. **Verify Username and Password:** Ensure that the username (billing) and password (raccoon) you provided are indeed correct and have been added to the authentication backend (SASL database or other mechanisms).
2. **Check SASL Configuration:** Verify that Postfix is correctly set up to use SASL for authentication. Review your `/etc/postfix/main.cf` to ensure `smtpd_sasl_auth_enable = yes` is set and that the SASL paths and mechanisms are correctly defined.

3. **SASL Daemon Logs:** Check the logs for the SASL daemon (often Cyrus SASL). This might give more detailed error messages on why authentication is failing.
4. **Test with Different Authentication Mechanisms:** You can specify a specific authentication mechanism with swaks by using the `--auth-type` option. For instance, try `--auth-type PLAIN` or `--auth-type LOGIN` to see if one mechanism might be working over the others.
5. **Consider Using Full Email for Username:** Even though you mentioned that you want to use just the username without the domain, sometimes systems are configured to expect the full email address. It might be worth trying `billing@plan6.com` just to see if that resolves the issue, even if it's not the desired final configuration.
6. **Logs:** Always check the Postfix logs for more detailed error messages. The logs might give clues on what's going wrong, especially with authentication.

Remember, email and SMTP debugging can be a complex process given the multiple layers and components involved. Taking a systematic approach and going step by step will help you diagnose and solve the issue.



How do I check the long of postfix again



To check the logs of Postfix, you typically look in the `/var/log/mail.log` or `/var/log/maillog` file, depending on your distribution and how logging is set up.

Here's how you can do it:

1. **Using `cat` command:** This will display the entire content of the file. It's useful if the file is not too long.

```
bash
cat /var/log/mail.log
```

2. **Using `tail` command:** This displays the last few lines of the file. It's especially useful to check recent activities.

```
bash
tail -n 100 /var/log/mail.log
```

Here `-n 100` will show the last 100 lines. You can adjust the number based on your needs.

3. **Using `grep` command:** This allows you to search for specific keywords or patterns. For instance, if you want to check for all authentication failures:

```
bash
grep "authentication failure" /var/log/mail.log
```

4. **Using `less` or `more` commands:** These commands allow you to view the log files in a paginated manner, so you can scroll up and down.

```
bash
less /var/log/mail.log
```

or

```
bash
more /var/log/mail.log
```

To exit from `less` or `more`, you can press `q`.

Make sure you have the necessary permissions to read the log file. If you encounter permission issues, you might need to use `sudo` with the above commands. For example:

```
bash
sudo tail -n 100 /var/log/mail.log
```



Does this have any additional helpful details? Sep 4 03:32:07 plan6 postfix/submission/smtpd[23648]: connect from ool-ad02faee.dyn.optonline.net[173.2.250.238] Sep 4 03:32:07 plan6 postfix/submission/smtpd[23648]: disconnect from ool-ad02faee.dyn.optonline.net[173.2.250.238] ehlo=1 mail=0/1 quit=1 commands=2/3 Sep 4 03:32:55 plan6 postfix/submission/smtpd[23648]: connect from ool-ad02faee.dyn.optonline.net[173.2.250.238] Sep 4 03:32:55 plan6 postfix/submission/smtpd[23648]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL CRAM-MD5 authentication failed: authentication failure Sep 4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: SASL authentication failure: unable to canonify user and get auxprops Sep 4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL DIGEST-MD5 authentication failed: authentication failure Sep 4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: SASL authentication failure: Password verification failed Sep 4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL PLAIN authentication failed: authentication failure Sep 4 03:32:56 plan6 postfix/submission/smtpd[23648]: disconnect from ool-ad02faee.dyn.optonline.net[173.2.250.238] ehlo=2 starttls=1 auth=0/4 quit=1 commands=4/8 Sep 4 03:36:02 plan6 postfix/anvil[23617]: statistics: max connection rate 3/60s for (submission:173.2.250.238) at Sep 4 03:29:53 Sep 4 03:36:02 plan6 postfix/anvil[23617]: statistics: max connection count 1 for (submission:173.2.250.238) at Sep 4 03:26:02 Sep 4 03:36:02 plan6 postfix/anvil[23617]: statistics: max cache size 2 at Sep 4 03:28:24



Let's analyze the logs:

1. The connection is made to the Postfix submission service.

```
less
Sep  4 03:32:07 plan6 postfix/submission/smtpd[23648]: connect from ool-ad02faee.dyn.optonline.net[173.2.250.238]
```

2. After the connection, there is an immediate disconnection without any mail transactions.

```
less
Sep  4 03:32:07 plan6 postfix/submission/smtpd[23648]: disconnect from ool-ad02faee.dyn.optonline.net[173.2.250.238]
```

3. Another connection is attempted:

```
less
Sep  4 03:32:55 plan6 postfix/submission/smtpd[23648]: connect from ool-ad02faee.dyn.optonline.net[173.2.250.238]
```

4. Multiple authentication failures:

- CRAM-MD5 authentication failed.
- Unable to canonify the user and get auxiliary properties. This can be related to how SASL is set up and can be a hint that the SASL system can't determine necessary user data. This could be due to issues in the user database or the lookup mechanism.
- DIGEST-MD5, LOGIN, and PLAIN authentications also failed.

```
less
Sep  4 03:32:55 plan6 postfix/submission/smtpd[23648]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL CRAM-MD5 authentication failed: authentication failure
Sep  4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: SASL authentication failure: unable to canonify user and get auxprops
Sep  4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL DIGEST-MD5 authentication failed: authentication failure
Sep  4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL LOGIN authentication failed: authentication failure
Sep  4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: SASL authentication failure: Password verification failed
Sep  4 03:32:56 plan6 postfix/submission/smtpd[23648]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL PLAIN authentication failed: authentication failure
```

5. The connection ends:

```
less
Sep  4 03:32:56 plan6 postfix/submission/smtpd[23648]: disconnect from ool-ad02faee.dyn.optonline.net[173.2.250.238]
```

6. Some statistical logs about the connections from the address 173.2.250.238:

```
less
Sep  4 03:36:02 plan6 postfix/anvil[23617]: statistics: max connection rate 3/60s for (submission:173.2.250.238) at Sep 4 03:29:53
Sep  4 03:36:02 plan6 postfix/anvil[23617]: statistics: max connection count 1 for (submission:173.2.250.238) at Sep 4 03:26:02
Sep  4 03:36:02 plan6 postfix/anvil[23617]: statistics: max cache size 2 at Sep  4 03:28:24
```

## Conclusions:

- There's an authentication problem. Given that multiple authentication methods are failing, it might be more related to user credentials or the way the user information is stored and retrieved than the actual authentication methods.
- The error "unable to canonify user and get auxprops" suggests a deeper issue with the SASL setup. It can be related to user data retrieval or mapping.

### Next Steps:

- Check the SASL configuration.
- Verify if the user `billing` exists in the user database you're using for authentication.
- Ensure that the password is stored correctly. If using hashed passwords, ensure the correct hashing mechanism is used.
- Test authentication locally on the server if possible.
- Check for more detailed errors in SASL or system logs. SASL might log more specific details elsewhere.



How does postfix know about the username/password table?

-----



Postfix doesn't handle authentication by itself for SMTP transactions. Instead, it offloads this task to the Cyrus Simple Authentication and Security Layer (SASL) framework. The actual storage and mechanism for authentication can vary, but it's commonly backed by one of several mechanisms:

1. **PAM (Pluggable Authentication Modules):** This uses the system's authentication mechanisms, effectively allowing users to authenticate against their Unix user/pass.
2. **sasldb:** This is a simple database mechanism provided by the Cyrus SASL library. It's useful for setups where you don't want to rely on system accounts.
3. **LDAP, SQL, etc.:** For larger setups or more dynamic systems, Postfix/SASL can authenticate against an LDAP directory, an SQL database, or other backends.

Here's a general overview of how this is set up:

1. **SASL Configuration:** In the Postfix `main.cf` configuration file, you'll have entries that determine how SASL is used:

```
makefile
smtpd_sasl_auth_enable = yes
smtpd_sasl_security_options = noanonymous
smtpd_sasl_path = smtpd
```

- The `smtpd_sasl_path` might point to a directory or specific configuration related to SASL. This will correspond to a configuration in the SASL library's setup.
2. **SASL Backend Configuration:** In the configuration referenced by `smtpd_sasl_path`, you'll define the mechanism to be used (e.g., `pwcheck_method: auxprop` or `pwcheck_method: sasldb`) and any necessary information for that backend, like database connection details for SQL backends.
  3. **Username/Password Storage:** Depending on the backend:
    - **For PAM:** It'll use system accounts, so adding a user to the system would effectively add them to the email system.
    - **For sasldb:** You'd use the `saslpasswd2` command to manage usernames and passwords.
    - **For LDAP/SQL:** You'd have entries in your directory or rows in your database that represent users and their associated passwords.

To figure out how your Postfix setup is configured to handle SASL authentication, you'd start by looking at the `main.cf` file and tracing from there to the SASL configuration and the defined backend. That would give you insight into where and how usernames and passwords are managed.

-----



```
# enable SASL authentication smtp_sasl_auth_enable = yes # disallow methods that allow anonymous authentication. smtp_sasl_security_options =
noanonymous # where to find sasl_passwd smtp_sasl_password_maps = hash:/etc/postfix/sasl_passwd # Enable STARTTLS encryption smtp_use_tls =
yes # where to find CA certificates smtp_tls_CAfile = /etc/ssl/certs/ca-certificates.crt I see this is missing the d
```

-----



The configuration you provided is for the Postfix SMTP **client**. It's used when Postfix is sending mail to another server and needs to authenticate to that server. Here's a breakdown of what each of those lines mean:

1. **smtp\_sasl\_auth\_enable = yes:** This enables authentication for the Postfix SMTP client.

2. **smtp\_sasl\_security\_options = noanonymous**: This ensures the Postfix SMTP client doesn't use any authentication methods that might result in anonymous logins.
3. **smtp\_sasl\_password\_maps = hash:/etc/postfix/sasl\_passwd**: This is where the Postfix SMTP client will look up the username and password to use when authenticating to another server. The file `/etc/postfix/sasl_passwd` will have entries that map server names to username/password pairs.
4. **smtp\_use\_tls = yes**: This ensures the Postfix SMTP client tries to use TLS encryption when connecting to another server.
5. **smtp\_tls\_CAfile = /etc/ssl/certs/ca-certificates.crt**: This is the location of trusted CA certificates which the Postfix SMTP client will use to verify the SSL certificate of the server it is connecting to.

You mentioned that it's missing the "d". If you were expecting configuration for the Postfix SMTP **server** (the part of Postfix that accepts incoming connections and mail), then you'd be looking for lines that start with `smtpd_` instead of `smtp_`. For example:

- `smtpd_sasl_auth_enable`
- `smtpd_sasl_security_options`
- `smtpd_sasl_path`
- `smtpd_tls_cert_file`
- `smtpd_tls_key_file`

If you're trying to set up SASL authentication for incoming connections to your Postfix server, you need to ensure the `smtpd_`-related configurations are correctly set up.



But that's from `main.cf`, that should be server, right?



Yes, you're correct. The `main.cf` is the main configuration file for Postfix, and it includes settings for both the SMTP client (when Postfix sends mail) and the SMTP server (when Postfix receives mail). The prefixes `smtp_` and `smtpd_` determine which component the setting applies to:

- `smtp_`: Relates to the SMTP client settings (when Postfix sends mail to other servers).
- `smtpd_`: Relates to the SMTP server settings (when Postfix receives mail from clients or other servers).

The configuration you posted pertains to the Postfix SMTP client settings. For SASL authentication for the Postfix SMTP server (inbound connections), you would typically see something like:

```
bash
smtpd_sasl_auth_enable = yes
smtpd_sasl_type = dovecot
smtpd_sasl_path = private/auth
smtpd_sasl_local_domain = $myhostname
smtpd_sasl_security_options = noanonymous, noplaintext
smtpd_sasl_tls_security_options = $smtpd_sasl_security_options
smtpd_sasl_authenticated_header = yes
```

Additionally, you would need to set up an appropriate authentication service (like Dovecot) that works with Postfix to authenticate users.

If you are trying to set up SASL for inbound connections to your Postfix server, you need to check and ensure the `smtpd_`-related configurations are correctly set up and that you have an appropriate backend for authentication (often Dovecot).

If you already have these configurations in place and are still facing issues, please provide the related `smtpd_` configurations from your `main.cf` file, and I can help you further.



Do you see anything here that might be causing authentication issues (`main.cf`): See `/usr/share/postfix/main.cf.dist` for a commented, more complete version # Debian specific: Specifying a file name will cause the first # line of that file to be used as the name. The Debian default # is `/etc/mailname`.  
#myorigin = `/etc/mailname` smtpd\_banner = \$myhostname ESMTP \$mail\_name (Debian/GNU) biff = no # appending .domain is the MUA's job.  
append\_dot\_mydomain = no # Uncomment the next line to generate "delayed mail" warnings #delay\_warning\_time = 4h readmedirectory = no # See [http://www.postfix.org/COMPATIBILITY\\_README.html](http://www.postfix.org/COMPATIBILITY_README.html) -- default to 2 on # fresh installs. compatibility\_level = 2 # TLS parameters  
smtpd\_tls\_cert\_file=/etc/letsencrypt/live/plan6.com/fullchain.pem smtpd\_tls\_key\_file=/etc/letsencrypt/live/plan6.com/privkey.pem  
smtpd\_tls\_security\_level=may smtpd\_tls\_CApath=/etc/ssl/certs smtpd\_tls\_security\_level=may smtpd\_tls\_session\_cache\_database =  
btree:\${data\_directory}/smtpd\_scache smtpd\_relay\_restrictions = permit\_mynetworks permit\_sasl\_authenticated defer\_unauth\_destination myhostname =  
plan6.com alias\_maps = hash:/etc/aliases alias\_database = hash:/etc/aliases myorigin = `/etc/mailname` mydestination = \$myhostname, plan6.com,  
localhost.com, localhost relayhost = mynetworks = 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128 mailbox\_size\_limit = 0 recipient\_delimiter = + inet\_interfaces =

all inet\_protocols = all # enable SASL authentication smtp\_sasl\_auth\_enable = yes # disallow methods that allow anonymous authentication.  
smtp\_sasl\_security\_options = noanonymous # where to find sasl\_passwd smtp\_sasl\_password\_maps = hash:/etc/postfix/sasl\_passwd # Enable  
STARTTLS encryption smtp\_use\_tls = yes # where to find CA certificates smtp\_tls\_CAfile = /etc/ssl/certs/ca-certificates.crt



From the provided configuration (main.cf), you have set up the SMTP client-side for SASL authentication (for outgoing emails from your server to other servers) but not for the SMTP server side (for incoming emails to your server). If you want SASL authentication to work for incoming emails (clients connecting to your server to send emails), you need to set up the smtpd\_ related configurations for SASL.

Here's what you need to add or modify to enable SASL authentication for incoming connections:

**1. Enable SASL Authentication:**

```
bash
smtpd_sasl_auth_enable = yes
```

**2. Specify the SASL type:** If you're using Dovecot for authentication, add:

```
bash
smtpd_sasl_type = dovecot
smtpd_sasl_path = private/auth
```

**3. Set up SASL Security Options:** Disallow methods that allow anonymous authentication. You can also prevent plain text logins if not using TLS.

```
bash
smtpd_sasl_security_options = noanonymous
```

**4. Ensure permit\_sasl\_authenticated is in your smtpd\_relay\_restrictions:** This permits clients that authenticate successfully to relay mail. You've already got this set:

```
bash
smtpd_relay_restrictions = permit_mynetworks permit_sasl_authenticated defer_unauth_destination
```

Make sure you have the appropriate backend for authentication set up. Often, Postfix is used in conjunction with Dovecot for authentication. If you're using Dovecot, ensure it's correctly configured to provide SASL authentication for Postfix.

Lastly, after you make these changes, always remember to reload or restart the Postfix service:

```
bash
sudo systemctl reload postfix
```

If you're not using Dovecot or another specific SASL provider, you'll need to set up the necessary service and configurations accordingly.



How do I get it working with dove authentication



Using Dovecot for SASL authentication with Postfix is a common setup. Here's how you can set it up:

**1. Dovecot Configuration:**

First, make sure Dovecot is installed and properly set up to handle mail. Then, for Dovecot to provide SASL authentication, you need to ensure it has the right socket set up for Postfix to communicate with. The socket is typically a Unix socket.

In your Dovecot configuration (often found in /etc/dovecot/dovecot.conf or in a file under /etc/dovecot/conf.d/), you should have something like:

```
bash
```



```
service auth {
    unix_listener /var/spool/postfix/private/auth {
        mode = 0660
        user = postfix
        group = postfix
    }
    # ... other settings if present
}
```

This creates a Unix socket at `/var/spool/postfix/private/auth` that Postfix can communicate with.

## 2. Postfix Configuration:

Edit your Postfix main configuration (often `/etc/postfix/main.cf`) to set up the correct parameters for Dovecot SASL:

```
bash
# SASL support for server-side authentication
smtpd_sasl_auth_enable = yes
smtpd_sasl_type = dovecot
smtpd_sasl_path = private/auth
smtpd_sasl_security_options = noanonymous
smtpd_sasl_local_domain = $myhostname
smtpd_sasl_authenticated_header = yes
```

Ensure that `permit_sasl_authenticated` is included in your `smtpd_relay_restrictions` or `smtpd_recipient_restrictions` as it allows authenticated users to send email:

```
bash
smtpd_relay_restrictions = permit_mynetworks permit_sasl_authenticated defer_unauth_destination
```

## 3. Restart Services:

After making the changes, you'll need to reload or restart both Dovecot and Postfix for the changes to take effect:

```
bash
sudo systemctl restart dovecot
sudo systemctl restart postfix
```

## 4. Testing:

You should test the setup to ensure everything is working correctly. You can do this by trying to send an email through your server using an email client or utilities like `swaks` and observing the logs for any errors or issues.

## 5. Monitor Logs:

Always keep an eye on the logs during and after the setup. Both Postfix and Dovecot provide detailed logs that can help troubleshoot any issues.

For Postfix:

```
bash
sudo tail -f /var/log/mail.log
```

For Dovecot (depending on setup):

```
bash
sudo tail -f /var/log/dovecot.log
```

This is a basic setup, and you may need additional configurations based on your requirements like virtual users, quotas, etc. It's essential to ensure your setup is secure and doesn't act as an open relay.



You:

`smtpd_sasl_type = ....` I don't have that what are my options for authentication type? Do I need an authentication type?



ChatGPT:

The `smtpd_sasl_type` parameter in Postfix specifies the SASL (Simple Authentication and Security Layer) server implementation to use for authentication. By default, this is set to `cyrus` which uses the Cyrus SASL library.

Here are the common options for `smtpd_sasl_type`:

1. `cyrus`: This uses the Cyrus SASL library. If you choose this, you need the Cyrus SASL software to be installed and properly configured. It's a bit more complex and may offer more SASL mechanisms and integrations.
2. `dovecot`: This uses Dovecot's authentication server. When you use Dovecot for SASL authentication, you typically set this to `dovecot`.

If you're using Dovecot for authentication (as in the prior guidance), you should set:

```
makefile
smtpd_sasl_type = dovecot
```

If the parameter is missing in your `main.cf`, you can simply add it.

You don't always need to set an authentication type if you're happy with the default (`cyrus`). However, if you're using Dovecot for authentication, it's essential to specify `dovecot` as the type to make sure Postfix communicates correctly with Dovecot for SASL authentication.

-----



You:

How can I see if i have the Cyrus SASL

-----



ChatGPT:

To check if you have the Cyrus SASL library installed on a Debian/Ubuntu system, you can use the following command:

```
bash
dpkg -l | grep sasl2-bin
```

On a Red Hat/Fedora/CentOS system, you can use:

```
bash
rpm -q cyrus-sasl
```

If the command returns package details, it means the Cyrus SASL library is installed. If it doesn't return anything or gives an error like "package not installed," then it likely isn't installed.

If you're using another type of system, the package management command might vary.

Additionally, to check the SASL plugins available, you can list the contents of the SASL plugin directory:

```
bash
ls /usr/lib/sasl2
```

This will show the SASL mechanisms supported by your Cyrus SASL installation.

-----



You:

Oh I don't have cyrus, can i install it?

-----



ChatGPT:

Yes, you can install the Cyrus SASL library and its utilities. The process depends on the operating system you are using:

### For Debian/Ubuntu:

You can install the necessary packages with:

```
bash
sudo apt update
sudo apt install sasl2-bin libsasl2-2 libsasl2-modules
```

### For Red Hat/Fedora/CentOS:

```
bash
sudo yum install cyrus-sasl
```

or if you're using the newer dnf package manager:

```
bash
sudo dnf install cyrus-sasl
```

After installing, you might want to configure Postfix to use Cyrus SASL for authentication, as discussed earlier. You'll adjust your `main.cf` to set the `smtpd_sasl_type` to `cyrus`, and then further configure based on your requirements.

Make sure to restart Postfix after making any changes:

```
bash
sudo service postfix restart
```

If you're specifically aiming to integrate with Dovecot for authentication, then you might not need Cyrus SASL. Instead, you would set up Dovecot as the SASL authentication service for Postfix. The choice depends on your specific requirements and desired setup.

-----



You:

Oh I do have cyrus. How do i specify cyrus as a authentication type here: `swaks --to twcookson@gmail.com --from billing@plan6.com --server plan6.com:587 --auth LOGIN --tls`

-----



ChatGPT:

If you have Cyrus SASL installed and you've set up Postfix to use it for authentication, then `swaks` (the SMTP test tool) will automatically use the appropriate method when you use the `--auth` flag without specifying a specific method.

However, to be clear, the `--auth` option in `swaks` does not define which SASL library (like Cyrus) to use. Instead, it specifies the SASL mechanism (like PLAIN, LOGIN, CRAM-MD5, etc.). When you specify `--auth LOGIN`, you're asking `swaks` to use the LOGIN mechanism for authentication.

To use Cyrus SASL with `swaks`, you just need to ensure that:

1. Postfix is set up to use Cyrus for SASL.
2. The appropriate Cyrus SASL mechanisms are enabled and configured on the server side.

Then, when you use `swaks` with the `--auth` flag, it will interact with the server's Cyrus SASL setup.

So, if you have everything set up correctly, your `swaks` command remains the same:

```
bash
swaks --to twcookson@gmail.com --from billing@plan6.com --server plan6.com:587 --auth LOGIN --tls
```

Remember, the key is the server-side configuration. The `swaks` tool just acts as a client trying to authenticate, and the mechanism it uses (LOGIN in this case) should be supported and properly configured on the server side.

-----



You:

How do I make sure cyrus is working correctly with postfix?

-----



ChatGPT:

Ensuring that Cyrus SASL is correctly integrated with Postfix involves multiple steps. Here's a general checklist to make sure Cyrus is working correctly with Postfix:

1. **Installation of Packages:** Ensure that both Postfix and Cyrus SASL libraries are installed. On Debian/Ubuntu systems, you'd typically install the following packages:

```
bash
sudo apt-get install postfix libsasl2-2 sasl2-bin libsasl2-modules
```

2. **Postfix Main Configuration:**

- Open `/etc/postfix/main.cf`.
- Ensure the following lines are present:

```
makefile
smtpd_sasl_auth_enable = yes
smtpd_sasl_security_options = noanonymous
smtpd_sasl_local_domain = $myhostname
broken_sasl_auth_clients = yes
```

3. **SASL Daemon Configuration:**

- Edit the file `/etc/default/saslauthd`:

```
bash
START=yes
```

- Choose the authentication mechanism you want for Cyrus. One common mechanism is to authenticate against the system's user accounts. If so, set the mechanism in `/etc/default/saslauthd` to:

```
makefile
MECHANISMS="pam"
```

4. **Postfix and saslauthd Communication:**

- Create a directory for the saslauthd mux socket and give Postfix permission:

```
bash
sudo mkdir -p /var/spool/postfix/var/run/saslauthd
sudo chown root:sasl /var/spool/postfix/var/run/saslauthd
sudo usermod -a -G sasl postfix
```

- Update the saslauthd configuration to use this directory. Edit the file `/etc/default/saslauthd` and set:

```
CSS
OPTIONS="-c -m /var/spool/postfix/var/run/saslauthd"
```

5. **Restart Services:**

- After making changes, ensure you restart the relevant services:

```
bash
sudo service saslauthd restart
sudo service postfix restart
```

6. **Testing:**

- Test saslauthd directly:

```
bash
testsaslauthd -u username -p password
```

This command should return `0: OK "Success."` if authentication is successful.

- Test SMTP authentication using `swaks` or another email client.

7. **Secure Your Server:** Before deploying this setup, ensure that you:

- Only allow secure (TLS/SSL) connections if passwords are being transmitted.
- Limit the IPs or networks that can use your server to send email.
- Regularly monitor and analyze your mail logs for suspicious activity.

Remember, the specifics of integrating Cyrus SASL with Postfix can vary based on your needs and the specifics of your server setup. Always refer to the official documentation and consider testing on a non-production server first.



You:

I'm missing testsaslauthd



ChatGPT:

If you're missing the `testsaslauthd` tool, it's usually a part of the `sasl2-bin` package, which provides utilities for Cyrus SASL.

1. **Installation:** Ensure you have the package installed. On Debian/Ubuntu systems, you'd run:

```
bash
sudo apt-get install sasl2-bin
```

2. **Location:** Sometimes, the issue isn't that the tool isn't installed but that it's not in your system's PATH. You can try locating it with:

```
bash
which testsaslauthd
```

Or search for it on your system using:

```
bash
sudo find / -name testsaslauthd 2>/dev/null
```

If it's found but not in your PATH, you can run it using the complete path, e.g., `/path/to/testsaslauthd`.

3. **Manual Installation:** If, for some reason, you cannot locate it via the package manager, you might need to compile Cyrus SASL from source and install it manually. This is a more advanced option and typically isn't necessary if you're using a standard package manager on a common Linux distribution.
4. **Alternative:** If you're unable to get `testsaslauthd` running and just want to test SASL authentication with Postfix, you can do so by trying to send an email with a client that requires authentication, such as `swaks`, `telnet`, or a standard email client.

Always ensure that you are referring to the official documentation or trusted guides when making configuration changes or installing software, especially on production systems.



You:

tom@plan6:~\$ /usr/sbin/testsaslauthd -u billing -p raccoon connect() : Permission denied 0: tom@plan6:~\$ WHyyyyyyyy



ChatGPT:

The "Permission denied" error when running `testsaslauthd` often occurs because the socket it's trying to connect to (usually located at `/var/run/saslauthd`) requires elevated privileges.

Try running the command with `sudo`:

```
bash
sudo /usr/sbin/testsaslauthd -u billing -p raccoon
```

By doing so, you'll elevate your permissions and should be able to test the authentication without running into permission issues. If the authentication is correct, it should return `0: OK "Success."`. If there's an issue with authentication, it will provide a relevant error message.



You:

0: tom@plan6:~\$ sudo /usr/sbin/testsaslauthd -u billing -p raccoon 0: NO "authentication failed"



The "authentication failed" message indicates that the authentication attempt using the provided credentials (billing and raccoon) did not succeed. There could be several reasons for this:

1. **Incorrect credentials:** Double-check to ensure that the username (billing) and password (raccoon) are correct.
2. **Configuration Issues:** There might be misconfigurations in your SASL or Postfix setup.
3. **Backend issues:** Cyrus SASL can utilize various authentication mechanisms, such as PAM, LDAP, or SQL databases. The issue could be related to the backend service not working correctly or not being correctly configured.

Here are some steps to help diagnose the issue:

1. **Check the logs:** Look at the logs to see if there's any additional information on why the authentication failed.

```
bash
sudo tail -n 50 /var/log/auth.log
```

This will show you the last 50 lines of the auth log, which may contain errors or warnings that can provide more insight.

2. **Check SASL Configuration:** Ensure that your SASL configuration (/etc/postfix/sasl/smtpd.conf) is correctly set up. For example, if you're using PAM as your mechanism, it should look something like:

```
makefile
pwcheck_method: saslauthd
mech_list: PLAIN LOGIN
```

3. **Check saslauthd service:** Ensure the saslauthd service is running:

```
bash
sudo service saslauthd status
```

If it's not running, start it:

```
bash
sudo service saslauthd start
```

4. **Test other users:** If you have other users set up, test them with testsaslauthd to see if it's an issue specific to the billing user or a more general issue.

If these steps don't reveal the problem, you might have to dig deeper into your system's authentication mechanisms. How are you storing user credentials? Is it a local system account, or are you using an external mechanism like a database or LDAP? The solution might depend on that answer.



You:

```
16 Sep 4 04:44:04 plan6 saslauthd[27616]: : listening on socket: /var/run/saslauthd/mux Sep 4 04:44:04 plan6 sudo: pam_unix(sudo:session): session
closed for user root Sep 4 04:44:45 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ; COMMAND=/usr/bin/tail -n 40 /var/log/mail.log Sep 4
04:44:45 plan6 sudo: pam_unix(sudo:session): session opened for user root(uid=0) by tom(uid=1000) Sep 4 04:44:45 plan6 sudo:
pam_unix(sudo:session): session closed for user root Sep 4 04:45:34 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ;
COMMAND=/usr/bin/apt-get install sasl2-bin Sep 4 04:45:35 plan6 sudo: pam_unix(sudo:session): session opened for user root(uid=0) by tom(uid=1000)
Sep 4 04:45:37 plan6 sudo: pam_unix(sudo:session): session closed for user root Sep 4 04:45:48 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ;
USER=root ; COMMAND=/usr/bin/find / -name testsaslauthd Sep 4 04:45:48 plan6 sudo: pam_unix(sudo:session): session opened for user root(uid=0) by
tom(uid=1000) Sep 4 04:45:52 plan6 sudo: pam_unix(sudo:session): session closed for user root Sep 4 04:46:42 plan6 sudo: tom : TTY=pts/0 ;
PWD=/home/tom ; USER=root ; COMMAND=/usr/sbin/testsaslauthd -u billing -p raccoon Sep 4 04:46:42 plan6 sudo: pam_unix(sudo:session): session
opened for user root(uid=0) by tom(uid=1000) Sep 4 04:46:42 plan6 saslauthd[27616]: pam_unix(imap:auth): check pass; user unknown Sep 4 04:46:42
plan6 saslauthd[27616]: pam_unix(imap:auth): authentication failure; logname= uid=0 euid=0 tty= ruser= rhost= Sep 4 04:46:44 plan6 saslauthd[27616]:
DEBUG: auth_pam: pam_authenticate failed: Authentication failure Sep 4 04:46:44 plan6 saslauthd[27616]: : auth failure: [user=billing] [service=imap]
[realm=] [mech=pam] [reason=PAM auth error] Sep 4 04:46:44 plan6 sudo: pam_unix(sudo:session): session closed for user root Sep 4 04:47:15 plan6
sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ; COMMAND=/usr/bin/tail -n 50 /var/log/auth.log Sep 4 04:47:15 plan6 sudo:
pam_unix(sudo:session): session opened for user root(uid=0) by tom(uid=1000)
```



From the logs you've provided, we can infer a few things:

1. `saslauthd` is indeed running and processing authentication requests.
2. `saslauthd` is using PAM (Pluggable Authentication Modules) for authentication.
3. When attempting to authenticate the user `billing`, the system responds with "user unknown" for the `imap` service, which suggests that this user isn't recognized by PAM for the specified service.

Here are some steps to troubleshoot:

1. **User Existence:** Make sure the user `billing` exists on the system. You can check this by examining `/etc/passwd`:

```
bash
grep billing /etc/passwd
```

If the user doesn't exist, you may need to create it or ensure you're trying to authenticate with the right username.

2. **PAM Configuration:** Since `saslauthd` is using PAM for authentication, check the PAM configuration for the `imap` service. You might find relevant configurations in `/etc/pam.d/`. Look for a file named `imap` or similar.

Examine the file to see if there's any special configuration or restrictions. A common setup might point to system authentication, like:

```
swift
auth    required      pam_unix.so
account required      pam_unix.so
```

If the file isn't there, you might need to create one with a standard configuration or make sure `saslauthd` is looking in the right place.

3. **SASL Configuration:** Ensure your SASL configuration for Postfix is correct. This would typically be in `/etc/postfix/sasl/smtpd.conf`. Make sure it's set up to use `saslauthd` and the right mechanisms:

```
makefile
pwcheck_method: saslauthd
mech_list: PLAIN LOGIN
```

4. **Dovecot:** If you're using Dovecot as an IMAP server, you might want to integrate SASL authentication directly with Dovecot, instead of using PAM through `saslauthd`. This would be more efficient and avoid possible PAM configuration complexities. You can do this by modifying the Postfix configuration to use Dovecot's SASL implementation.

After making any changes, always remember to restart the relevant services (`postfix`, `saslauthd`, etc.) to apply the changes.

-----



```
tom@plan6:~$ grep billing /etc/passwd
tom@plan6:~$
```

-----



It seems that the user `billing` does not exist on your system, which explains the authentication failure.

If `billing` is supposed to be a mailbox user, you have a few choices:

1. **System User:** You can create `billing` as a system user with a home directory where you will store the mail. This approach ties mail accounts to system accounts, which might not be what you want for a larger mail setup or for security reasons.

To create the user:

```
bash
sudo useradd -m billing
sudo passwd billing
```

This would create a home directory for the user and allow you to set a password for the account.

2. **Virtual Users with Dovecot:** If you are using Dovecot for IMAP, you can set up virtual mailboxes without creating system users. This is a more scalable approach and decouples mail accounts from system accounts. It requires a bit more configuration, including setting up a virtual user list, and specifying mailbox locations. If you're going this route, you'll also need to integrate Postfix with Dovecot for SASL authentication.
3. **Other Authentication Backends:** `saslauthd` and Cyrus SASL support other authentication backends beyond PAM and shadow passwords. You could set up a database, for instance, to handle mail user authentication. This offers scalability and flexibility, but also requires additional setup.

Given the current state of your configuration and the nature of your problem, if `billing` is meant to be a mail user, the easiest immediate solution is to create it as a system user (as in option #1). If you're planning for a more complex mail setup in the future, you might want to look into options #2 or #3.



You:

```
tom@plan6:~$ sudo /usr/sbin/testsaslauthd -u billing -p raccoon 0: OK "Success."
```



ChatGPT:

Great, it looks like you've resolved the authentication issue for the `billing` user, and `testsaslauthd` is now indicating a successful authentication!

Now, if you retry the email test (with `swaks` or any other method you prefer), and ensure that Postfix is configured to use `saslauthd` for SASL authentication, you should be able to send email without facing authentication errors.

Always make sure to keep the authentication details secure and don't share passwords in logs or in other public spaces.



You:

```
Still fails authentication in mail log: 3.5.18, configuration /etc/postfix Sep 4 04:50:34 plan6 postfix/submission/smtpd[28298]: connect from ool-ad02faee.dyn.optonline.net[173.2.250.238] Sep 4 04:50:35 plan6 postfix/submission/smtpd[28298]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL LOGIN authentication failed: authentication failure Sep 4 04:50:35 plan6 postfix/submission/smtpd[28298]: disconnect from ool-ad02faee.dyn.optonline.net[173.2.250.238] ehlo=2 starttls=1 auth=0/1 quit=1 commands=4/5 tom@plan6:~$ Auth log: p 4 04:50:24 plan6 saslauthd[28293]: : listening on socket: /var/run/saslauthd/mux Sep 4 04:50:24 plan6 sudo: pam_unix(sudo:session): session closed for user root Sep 4 04:50:49 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ; COMMAND=/usr/sbin/testsaslauthd -u billing -p raccoon Sep 4 04:50:49 plan6 sudo: pam_unix(sudo:session): session opened for user root(uid=0) by tom(uid=1000) Sep 4 04:50:50 plan6 sudo: pam_unix(sudo:session): session closed for user root Sep 4 04:51:05 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ; COMMAND=/usr/bin/tail -n 50 /var/log/auth.log Sep 4 04:51:05 plan6 sudo: pam_unix(sudo:session): session opened for user root(uid=0) by tom(uid=1000) Sep 4 04:51:05 plan6 sudo: pam_unix(sudo:session): session closed for user root Sep 4 04:52:13 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ; COMMAND=/usr/bin/tail -n 40 /var/log/mail.log Sep 4 04:52:13 plan6 sudo: pam_unix(sudo:session): session opened for user root(uid=0) by tom(uid=1000) Sep 4 04:52:13 plan6 sudo: pam_unix(sudo:session): session closed for user root Sep 4 04:52:48 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ; COMMAND=/usr/bin/tail -n 50 /var/log/auth.log From client: === Trying plan6.com:587... === Connected to plan6.com. <- 220 plan6.com ESMTP Postfix (Debian/GNU) -> EHLO laptoplin <- 250-plan6.com <- 250-PIPELINING <- 250-SIZE 10240000 <- 250-VRIFY <- 250-ETRN <- 250-STARTTLS <- 250-ENHANCEDSTATUSCODES <- 250-8BITMIME <- 250-DSN <- 250-SMTPUTF8 <- 250-CHUNKING -> STARTTLS <- 220 2.0.0 Ready to start TLS === TLS started with cipher TLSv1.3:TLS_AES_256_GCM_SHA384:256 === TLS no local certificate set === TLS peer DN="/CN=plan6.com" -> EHLO laptoplin <- 250-plan6.com <- 250-PIPELINING <- 250-SIZE 10240000 <- 250-VRIFY <- 250-ETRN <- 250-AUTH DIGEST-MD5 CRAM-MD5 NTLM LOGIN PLAIN <- 250-AUTH=DIGEST-MD5 CRAM-MD5 NTLM LOGIN PLAIN <- 250-ENHANCEDSTATUSCODES <- 250-8BITMIME <- 250-DSN <- 250-SMTPUTF8 <- 250-CHUNKING -> AUTH LOGIN <- 334 VXNlcm5hbWU6 -> YmlsbGluZw== <- 334 UGFzc3dvcmQ6 -> cmFjY29vbG== <-* 535 5.7.8 Error: authentication failed: authentication failure *** No authentication type succeeded -> QUIT <- 221 2.0.0 Bye === Connection closed with remote host.
```



ChatGPT:

Your server is correctly offering several authentication mechanisms (DIGEST-MD5, CRAM-MD5, NTLM, LOGIN, PLAIN). The client chose to use the LOGIN mechanism and failed.

Let's try to piece this together:

1. `testsaslauthd` worked correctly after some initial hitches, indicating that `saslauthd` is working.
2. Postfix is giving an authentication error which indicates the connection between Postfix and `saslauthd` might be the problem.

Here are steps to ensure Postfix is correctly configured to use `saslauthd`:

1. Ensure that `smtpd_sasl_type` is set to `cyrus` in your `main.cf`:

```
makefile
smtpd_sasl_type = cyrus
```

This tells Postfix to use Cyrus SASL for authentication.

2. Ensure the path for `saslauthd` is correctly set:

```
javascript
smtpd_sasl_path = /var/run/saslauthd
```



3. Confirm Postfix is using the correct SASL plugins:

The folder `/usr/lib/sasl2/` (or `/usr/lib64/sasl2/` on 64-bit systems) contains the plugins that Cyrus SASL (and thus Postfix) will use. There should be libraries like `liblogin.so`, `libplain.so`, etc., that correspond to the authentication mechanisms.

4. Ensure permissions:

Ensure that the Postfix user (often "postfix") has access to the `/var/run/saslauthd/` directory and its contents. If not, you might need to adjust permissions or group memberships.

You can add postfix to the sasl group:

```
bash
sudo adduser postfix sasl
```

Then restart the Postfix service:

```
bash
sudo systemctl restart postfix
```

5. Finally, try restarting both saslauthd and Postfix services to ensure that all configurations are loaded:

```
bash
sudo systemctl restart saslauthd postfix
```

After making these changes and ensuring configurations are loaded, retry your email authentication. If it fails again, the logs should hopefully provide a clearer indication of what's going wrong. If there's no change in the error, it could be an issue of Postfix not being able to communicate with saslauthd correctly or a misconfiguration in the SASL plugins/settings.



No change

-----



Alright, since there's no change after the steps provided, let's dig a little deeper. To pinpoint the issue, it's crucial to methodically eliminate potential causes.

1. **Double-check configuration files:** Ensure that all of the changes you made to the Postfix configuration files are saved and correct. Typos or small mistakes in these files can cause big issues.
2. **Postfix SASL Settings:** In the `/etc/postfix/main.cf` file, verify these settings:

```
makefile
smtpd_sasl_auth_enable = yes
smtpd_sasl_security_options = noanonymous
smtpd_sasl_local_domain = $myhostname
broken_sasl_auth_clients = yes
```

The `broken_sasl_auth_clients = yes` option allows for older email clients to connect, even if they do not follow the SASL protocol correctly.

3. **Postfix and Cyrus integration:** Ensure Postfix is looking for the Cyrus SASL socket in the correct place. The `smtpd_sasl_path` directive should point to the directory where saslauthd is creating its communication socket, typically `/var/run/saslauthd`. So, in `main.cf`:

```
javascript
smtpd_sasl_path = /var/run/saslauthd
```

4. **Apparmor or SELinux:** If you have Apparmor or SELinux enforcing, they could be blocking the connection between Postfix and saslauthd. Check the logs for these services to see if they are producing any blocks or errors related to Postfix or saslauthd.
5. **Logging:** Increase the logging verbosity of Postfix to get more detailed information:

In `/etc/postfix/master.cf`, look for the `smtpd` line and change `-v` at the end to `-vvv` (three 'v's for more verbosity).

Restart Postfix:

```
bash
sudo systemctl restart postfix
```

Then try authenticating again and check the logs. The extra verbose logs may provide more insight into where the breakdown is occurring.

6. **Manual SASL Test:** Try manually testing SASL from the command line. This can give a clearer error message:

```
bash
sudo /usr/sbin/saslauthd -a pam -c -m /var/spool/postfix/var/run/saslauthd -r -d
```

This starts saslauthd in debug mode. Now try to authenticate again using your mail client. The terminal running saslauthd should provide verbose output about the authentication attempt.

7. **Authentication Mechanism:** If you're sure that Postfix and saslauthd are communicating correctly, there may be an issue with the actual authentication mechanism (e.g., PAM). Ensure the PAM configuration is set up correctly for SMTP authentication.

After going through these steps, if you still encounter issues, please share any new or different log entries. They might contain the clues needed to understand and solve the problem.



You:

Add -vvv here? of the file, see the master(5) manual page (command: "man 5 master" or # on-line: <http://www.postfix.org/master.5.html>). ## Do not forget to execute "postfix reload" after editing this file. ## ===== #  
service type private unpriv chroot wakeup maxproc command + args # (yes) (yes) (no) (never) (100) #  
===== smtp inet n - y - - smtpd #smtp inet n - y - 1 postscreen  
#smtpd pass - - y - - smtpd #dnsblog unix - - y - 0 dnsblog #tlsproxy unix - - y - 0 tlsproxy submission inet n - y - - smtpd -o smtpd\_bind\_address=0.0.0.0 -o  
syslog\_name=postfix/submission -o smtpd\_tls\_security\_level=encrypt -o smtpd\_sasl\_auth\_enable=yes -o smtpd\_tls\_auth\_only=yes -o  
smtpd\_reject\_unlisted\_recipient=no -o smtpd\_client\_restrictions=\$mua\_client\_restrictions -o smtpd\_helo\_restrictions=\$mua\_helo\_restrictions -o  
smtpd\_sender\_restrictions=\$mua\_sender\_restrictions -o smtpd\_recipient\_restrictions=permit\_sasl\_authenticated,reject -o  
milter\_macro\_daemon\_name=ORIGINATING #smtps inet n - y - - smtpd # -o syslog\_name=postfix/smtps # -o smtpd\_tls\_wrappermode=yes # -o  
smtpd\_sasl\_auth\_enable=yes # -o smtpd\_reject\_unlisted\_recipient=no # -o smtpd\_client\_restrictions=\$mua\_client\_restrictions # -o  
smtpd\_helo\_restrictions=\$mua\_helo\_restrictions # -o smtpd\_sender\_restrictions=\$mua\_sender\_restrictions # -o smtpd\_recipient\_restrictions= # -o  
smtpd\_relay\_restrictions=permit\_sasl\_authenticated,reject # -o milter\_macro\_daemon\_name=ORIGINATING #628 inet n - y - - qmqpd pickup unix n - y 60  
1 pickup cleanup unix n - y 0 cleanup qmgr unix n - n 300 1 qmgr #qmgr unix n - n 300 1 oqmgr tlsmgr unix - - y 1000? 1 tlsmgr



ChatGPT:

Based on the provided portion of the /etc/postfix/master.cf file, you should add the -vvv to the smtpd line under submission. This will increase the verbosity of the logging for the smtpd process when it's handling submission requests.

Here's how you should modify the relevant portion of the file:

```
bash
submission inet n      -      y      -      -      smtpd -o smtpd_bind_address=0.0.0.0
-o syslog_name=postfix/submission
-o smtpd_tls_security_level=encrypt
-o smtpd_sasl_auth_enable=yes
-o smtpd_tls_auth_only=yes
-o smtpd_reject_unlisted_recipient=no
-o smtpd_client_restrictions=$mua_client_restrictions
-o smtpd_helo_restrictions=$mua_helo_restrictions
-o smtpd_sender_restrictions=$mua_sender_restrictions
-o smtpd_recipient_restrictions=
-o smtpd_relay_restrictions=permit_sasl_authenticated,reject
-o milter_macro_daemon_name=ORIGINATING
-v -v -v
```

Note the -v -v -v added at the end of the submission service block.

After making this change, remember to reload or restart Postfix:

```
bash
sudo systemctl reload postfix
```

Try the authentication again and then check your mail logs for more detailed information. This should provide more verbose output, which might help in diagnosing the problem.



You:

Sep 4 05:03:07 plan6 sudo: pam\_unix(sudo:session): session closed for user root Sep 4 05:03:10 plan6 sudo: tom : TTY=pts/0 ; PWD=/home/tom ; USER=root ; COMMAND=/usr/bin/systemctl restart saslauthd postfix Sep 4 05:03:10 plan6 sudo: pam\_unix(sudo:session): session opened for user root(uid=0) by tom(uid=1000) Sep 4 05:03:10 plan6 saslauthd[28961]: : master exited: 28961 Sep 4 05:03:10 plan6 saslauthd[29580]: : master pid is: 29580 Sep 4 05:03:10 plan6 saslauthd[29580]: : listening on socket: /var/run/saslauthd/mux Sep 4 05:03:13 plan6 sudo: pam\_unix(sudo:session): session closed for user root tom@plan6:~\$



The provided logs seem to be from the `auth.log` (or similar authentication-related log) which shows actions related to authentication mechanisms on the system, including `sudo` commands and the starting of the `saslauthd` service.

To diagnose the authentication issue with Postfix, you'll need to check the mail logs after you've added `-v -v -v` to the `master.cf` file for the submission service and restarted Postfix.

Please try to authenticate again (as you did before) and then provide the logs from `/var/log/mail.log` to see the verbose output from the Postfix `smtpd` process. This should give us a more detailed insight into why the authentication might be failing.



You:

There's a lot of data, how can I find useful info



When you have verbose logging enabled, the logs will indeed be more detailed, making it harder to spot the specific issue at hand. However, you can narrow down the search by filtering the logs based on the timestamp and key phrases.

Here's a step-by-step process to narrow down and find useful information:

#### 1. Timestamp:

- Every time you try to authenticate, note the exact time. The logs will have a timestamp, so you can search around the time you attempted the connection.

#### 2. Use `grep`:

- Use `grep` to filter out lines containing specific keywords. Given that you're troubleshooting SASL authentication, you can start by filtering lines that contain "SASL":

```
bash
grep "SASL" /var/log/mail.log
```

#### 3. Context Lines:

- Once you have located the lines of interest using `grep`, you can use `-A` and `-B` flags with `grep` to print the lines before and after the match. This can help provide context:

```
bash
grep -A 5 -B 5 "SASL" /var/log/mail.log
```

#### 4. Look for Errors and Warnings:

- Scan for words like `error`, `fail`, `warn`, or `cannot`. These are usually indicative of problems:

```
bash
grep -i "error\|fail\|warn\|cannot" /var/log/mail.log
```

#### 5. Review Logs from Client Side:

- If you have access to the mail client logs, it's beneficial to review those in parallel. It can give a clearer picture when you correlate both server and client logs.

#### 6. Limit the Amount of Log Data:

- If you know the exact time range you're interested in, you can use tools like `awk` to extract logs only from that range. For example, if you know the issue occurred around 05:03, you can do something like:

```
bash
awk '$3 ~ /^05:03:/ {print $0}' /var/log/mail.log
```

This command will only show logs from the time 05:03.

## 7. Documentation & Online Resources:

- If you spot an error message that you're unfamiliar with, it's a good idea to search for it online or refer to the official Postfix documentation. Sometimes others may have faced similar issues, and their solutions could provide clues.

Remember that with the verbose option `-v -v -v`, the Postfix logs will have a lot of information that may seem irrelevant. The trick is to filter out the noise and focus on lines that seem to be directly related to the authentication process.

If you have trouble interpreting the logs, feel free to share the relevant snippets here, and I'll help analyze them.

-----



```
grep "SASL" /var/log/mail.log
```

-----



You've provided a command that filters the mail log for lines containing "SASL". To proceed, you'd execute this command in your terminal. This will show you all entries in `/var/log/mail.log` that include the word "SASL", which can be helpful in identifying issues related to SASL authentication.

Once you have the output of this command, you can review it to look for errors or anomalies related to the authentication attempts. For better understanding, share the output (or relevant portions of it) here, and I can help you analyze and troubleshoot further.

Just remember when sharing logs:

- Redact any sensitive or personally identifiable information.
- Only share what is necessary to diagnose the problem, focusing on error messages or warnings.

-----



Hmmm... I dug through the log and found this line: `Sep 3 04:32:32 plan6 postfix/smtpd[7029]: warning: ool-ad02faee.dyn.optonline.net[173.2.250.238]: SASL LOGIN authentication failed: generic failure Sep 3 04:32:36 plan6 postfix/smtpd[7029]: warning: SASL authentication failure: cannot connect to saslauthd server: Too many levels of symbolic links` That seems important

-----



Yes, that's certainly an important clue. The error message:

```
arduino
SASL authentication failure: cannot connect to saslauthd server: Too many levels of symbolic links
```

suggests there's an issue with the way saslauthd socket files are being handled.

Here are steps to troubleshoot and resolve this issue:

### 1. Check the socket directory:

- The saslauthd service usually uses a UNIX socket located in `/var/run/saslauthd/` to communicate.
- Check the content of the directory and ensure there aren't any incorrect symbolic links:

```
bash
ls -l /var/run/saslauthd/
```

### 2. Correct permissions:

- Ensure that the Postfix user has permission to access the saslauthd mux socket.
- You might need to add the Postfix user to the sasl group or adjust the permissions on the `/var/run/saslauthd/` directory.

### 3. Configuration:

- In the Postfix configuration, ensure that `smtpd_sasl_path` is pointing to the correct location of the `saslauthd` mux. Typically, it should be set to something like:

```
makefile
smtpd_sasl_path = private/saslauthd/mux
```

- Ensure this path is consistent with where `saslauthd` is actually placing its socket.

#### 4. Restart Services:

- After making changes, ensure you restart both `saslauthd` and Postfix:

```
swift
sudo systemctl restart saslauthd
sudo systemctl restart postfix
```

#### 5. Check the `saslauthd` configuration:

- The `saslauthd` service might be configured to use a different mechanism or socket directory. You can check its configuration in `/etc/default/saslauthd` or similar location depending on your distribution. Ensure the `MECHANISMS` and `OPTIONS` are set correctly.

#### 6. Symbolic Links:

- The error suggests an issue with symbolic links. If you find any symbolic links in the `/var/run/saslauthd/` directory, ensure they point to valid locations and there's no loop or excessive levels of symbolic links.

After following these steps, try to authenticate again and see if the issue persists. If it does, you might want to consider looking deeper into the `saslauthd` and Postfix configurations or checking system logs for more detailed error messages.



```
tom@plan6:~$ sudo ls -l /var/run/saslauthd/ total 968 -rw----- 1 root root 0 Sep 4 05:03 cache.flock -rw----- 1 root root 986112 Sep 4 05:03 cache.mmap
srwxrwxrwx 1 root root 0 Sep 4 05:03 mux -rw----- 1 root root 0 Sep 4 05:03 mux.accept -rw----- 1 root root 6 Sep 4 05:03 saslauthd.pid tom@plan6:~$
```



The output you've provided looks generally fine in terms of file existence and permissions, especially with the mux socket being world-writable (which should allow any user, including the Postfix user, to access it).

Given that everything seems in order at the file system level, the error message of "Too many levels of symbolic links" is a bit puzzling, unless somewhere in the Postfix or SASL configuration there's a reference to a problematic symbolic link.

Let's try the following:

#### 1. Postfix Configuration: Check the Postfix configuration to ensure it's pointing to the right location for `saslauthd`.

- First, find out where Postfix is looking for the `saslauthd` mux with:

```
bash
postconf | grep smtpd_sasl_path
```

If it's not pointing to `/var/run/saslauthd/mux`, you'll need to set it:

```
bash
sudo postconf -e 'smtpd_sasl_path = /var/run/saslauthd/mux'
```

#### 2. Symlink Check: While the `ls` command you ran did not show any symbolic links in the `/var/run/saslauthd/` directory, the error you provided earlier is suggestive of an issue with symbolic links. Let's double-check:

- Check if `/var/run` is a symlink:

```
bash
ls -ld /var/run
```

- Check if `/var` itself is a symlink:

```
bash
ls -ld /var
```

3. **saslauthd Configuration:** The default settings for saslauthd are usually fine, but if someone has made changes, it's worth checking:

- Take a look at the /etc/default/saslauthd file (or its equivalent location on your distribution):

```
bash
cat /etc/default/saslauthd
```

Ensure that the OPTIONS line (if present) doesn't have a -m flag pointing to a different directory.

#### 4. Restart and Test:

- Restart the services:

```
bash
sudo systemctl restart saslauthd
sudo systemctl restart postfix
```

- Then, attempt to send an email using your client and see if the authentication error persists.

If, after all of this, the issue remains, it might be a deeper configuration issue or a more unusual system state, and further diagnostic steps would be required.

-----



You:

```
tom@plan6:~$ postconf | grep smtpd_sasl_path -bash: postconf: command not found tom@plan6:~$
```

-----



ChatGPT:

It seems like postconf is not in your PATH or it's not installed, which is a bit unusual because postconf is part of the Postfix package and is used to get and set Postfix configuration values.

Let's try the following:

#### 1. Find the postconf command:

```
bash
sudo find / -name postconf 2>/dev/null
```

This will search the entire filesystem for the postconf command and ignore any permission error messages.

If it finds the command, you can use the full path to execute it, like:

```
bash
sudo /path/to/postconf | grep smtpd_sasl_path
```

Replace /path/to/ with the actual path find gives you.

#### 2. Check if Postfix is properly installed:

If you can't find the postconf command, it might be worth ensuring Postfix is installed correctly.

Depending on your package manager:

For **APT** (Ubuntu, Debian):

```
bash
sudo apt-get install --reinstall postfix
```

For **YUM** (CentOS, RHEL):

```
bash
sudo yum reinstall postfix
```

#### 3. After ensuring Postfix is correctly installed, try the postconf command again to see if it's recognized:

```
bash
sudo postconf | grep smtpd_sasl_path
```

If you're still having issues after these steps, there might be deeper configuration or system issues at play.



```
-bash: postconf: command not found tom@plan6:~$ sudo find / -name postconf 2>/dev/null /snap/core/15925/usr/share/bash-completion/completions/postconf /snap/core/15511/usr/share/bash-completion/completions/postconf /snap/core20/2015/usr/share/bash-completion/completions/postconf /snap/core20/1974/usr/share/bash-completion/completions/postconf /usr/sbin/postconf tom@plan6:~$
```

-----



Ah, there it is. The `postconf` binary is located in `/usr/sbin/`, which might not be in the `PATH` for your user by default.

Use the full path to execute it:

```
bash
sudo /usr/sbin/postconf | grep smtpd_sasl_path
```

This should give you the configuration value for `smtpd_sasl_path` which will tell us where Postfix is trying to connect to `saslauthd`.



```
tom@plan6:~$ sudo /usr/sbin/postconf | grep smtpd_sasl_path /usr/sbin/postconf: warning: /etc/postfix/master.cf: undefined parameter: mua_sender_restrictions /usr/sbin/postconf: warning: /etc/postfix/master.cf: undefined parameter: mua_client_restrictions /usr/sbin/postconf: warning: /etc/postfix/master.cf: undefined parameter: mua_helo_restrictions smtpd_sasl_path = /var/run/saslauthd /usr/sbin/postconf: warning: /etc/postfix/master.cf: unused parameter: smtpd_bind_address=0.0.0.0 tom@plan6:~$
```

-----



The output confirms that Postfix is correctly configured to connect to `saslauthd` at `/var/run/saslauthd`. However, the warnings shown indicate potential configuration issues:

1. The parameters `mua_sender_restrictions`, `mua_client_restrictions`, and `mua_helo_restrictions` are mentioned in `master.cf` but are undefined.
2. There's an unused parameter `smtpd_bind_address=0.0.0.0`.

For the immediate SASL authentication issue, the relevant configuration (`smtpd_sasl_path`) seems correct, so it might not be a misconfiguration in `main.cf` or `master.cf` causing the problem.

Remember, we saw the error message `SASL authentication failure: cannot connect to saslauthd server: Too many levels of symbolic links`.

Here's a possible reason: The `saslauthd` might expect the path to be `/var/spool/postfix/var/run/saslauthd`, especially if Postfix is chrooted (which it often is by default).

To fix the issue:

1. Create a directory in the chroot for `saslauthd`:

```
bash
sudo mkdir -p /var/spool/postfix/var/run/saslauthd
```

2. Modify the `saslauthd` service file to use this new directory.

```
bash
sudo nano /etc/default/saslauthd
```

Change the line `OPTIONS="-c -m /var/run/saslauthd"` to `OPTIONS="-c -m /var/spool/postfix/var/run/saslauthd"`.

3. Restart `saslauthd`:

```
bash
```

```
sudo systemctl restart saslauthd
```

4. Make sure that Postfix has access to the new directory:

```
bash
sudo postconf -e 'smtpd_sasl_path = /var/spool/postfix/var/run/saslauthd'
sudo systemctl reload postfix
```

5. Try authenticating again.

If it still doesn't work, there might be deeper issues with the saslauthd configuration or potentially something else in the Postfix configuration, especially given those warnings. The warnings also indicate there might be some lines in `master.cf` that are misconfigured and should be addressed.



Lets remove everything to do with postfix

-----



Sure, if you want to remove Postfix and its associated configuration files from your system, you can do so using the package manager of your Linux distribution.

For Debian and Ubuntu-based systems:

1. **Uninstall Postfix:**

```
bash
sudo apt-get purge postfix
```

2. **Delete configuration files and directories:** Even after purging, some configuration files or directories may remain. You can manually remove them:

```
bash
sudo rm -rf /etc/postfix/
```

3. **Remove associated packages or dependencies (if desired):** When you installed Postfix, it may have installed additional packages or dependencies. To remove packages that were automatically installed to satisfy dependencies for other packages and are now no longer needed, you can use:

```
bash
sudo apt-get autoremove
```

For Red Hat or CentOS-based systems:

1. **Uninstall Postfix:**

```
bash
sudo yum remove postfix
```

2. **Delete configuration files and directories:**

```
bash
sudo rm -rf /etc/postfix/
```

3. Clean up any leftover dependencies:

```
bash
sudo yum autoremove
```

**Note:** Before removing any package or configuration, it's always a good practice to backup any data or configurations that you may need in the future. This ensures that if you ever decide to reinstall or if something goes wrong, you can restore your setup to its previous state.

-----





Explain to me how to install postfix with dovecot on debian



Certainly! Installing Postfix with Dovecot on a Debian system allows you to set up a mail server to receive and store emails and also provides mechanisms for authentication. Here's a step-by-step guide:

## 1. Update your System:

```
bash
sudo apt update && sudo apt upgrade -y
```

## 2. Install Postfix:

During the installation, you'll be prompted with some configuration choices:

- **General type of mail configuration:** Choose "Internet Site"
- **System mail name:** You can leave this as the default, which should be your server's fully qualified domain name (FQDN).

```
bash
sudo apt install postfix
```

## 3. Install Dovecot:

Install the Dovecot IMAP and POP3 server package, and also the Dovecot's Postfix integration:

```
bash
sudo apt install dovecot-core dovecot-imapd dovecot-pop3d dovecot-lmtpd
```

## 4. Configuring Postfix:

Edit the `main.cf` file:

```
bash
sudo nano /etc/postfix/main.cf
```

Add or modify the following lines:

```
plaintext
home_mailbox = Maildir/
smtpd_sasl_type = dovecot
smtpd_sasl_path = private/auth
smtpd_sasl_auth_enable = yes
smtpd_sasl_security_options = noanonymous
smtpd_sasl_local_domain = $myhostname
smtpd_recipient_restrictions = permit_mynetworks, permit_sasl_authenticated, reject_unauth_destination
```

## 5. Configuring Dovecot:

a. Configure the `10-auth.conf` file:

```
bash
sudo nano /etc/dovecot/conf.d/10-auth.conf
```

Find and modify the following lines:

```
plaintext
disable_plaintext_auth = no
auth_mechanisms = plain login
```

b. Configure the `10-mail.conf` file for mail storage:

```
bash
sudo nano /etc/dovecot/conf.d/10-mail.conf
```

Modify the following line:

```
plaintext
mail_location = maildir:~/Maildir
```

c. Configure the 10-master.conf file for service settings:

```
bash
sudo nano /etc/dovecot/conf.d/10-master.conf
```

Find the section for Postfix SMTP server and modify:

```
plaintext
unix_listener /var/spool/postfix/private/auth {
    mode = 0666
    user = postfix
    group = postfix
}
```

## 6. Restart Postfix and Dovecot:

```
bash
sudo systemctl restart postfix
sudo systemctl restart dovecot
```

## 7. Test:

Now, you can use an email client to test both IMAP and SMTP functions. For IMAP, you'll be connecting to Dovecot, and for SMTP, you'll be connecting to Postfix.

This guide provides a basic configuration to get you started. Depending on your requirements, you might want to add additional security measures (like SSL/TLS, SPF, DMARC, etc.), user quota management, virtual domains, spam filtering, and more.

Remember, running a mail server is a significant responsibility. Make sure to monitor and maintain it to prevent misuse and to ensure its security and reliability.



Can you help me find the Postfix SMTP server: #default\_process\_limit = 100 #default\_client\_limit = 1000 # Default VSZ (virtual memory size) limit for service processes. This is mainly # intended to catch and kill processes that leak memory before they eat up # everything. #default\_vsz\_limit = 256M # Login user is internally used by login processes. This is the most untrusted # user in Dovecot system. It shouldn't have access to anything at all. #default\_login\_user = dovenull # Internal user is used by unprivileged processes. It should be separate from # login user, so that login processes can't disturb other processes. #default\_internal\_user = dovecot service imap-login { inet\_listener imap { #port = 143 } inet\_listener imaps { #port = 993 #ssl = yes } # Number of connections to handle before starting a new process. Typically # the only useful values are 0 (unlimited) or 1. 1 is more secure, but 0 # is faster. <doc/wiki/LoginProcess.txt> #service\_count = 1 # Number of processes to always keep waiting for more connections. #process\_min\_avail = 0 # If you set service\_count=0, you probably need to grow this. #vsz\_limit = \$default\_vsz\_limit } Service pop3-login { inet\_listener pop3 { #port = 110 } inet\_listener pop3s { #port = 995 #ssl = yes } } service submission-login { inet\_listener submission { #port = 587 } } service lmtp { unix\_listener lmtp { #mode = 0666 } # Create inet listener only if you can't use the above UNIX socket #inet\_listener lmtp { # Avoid making LMTP visible for the entire internet #address = #port = # } } service imap { # Most of the memory goes to mmap()ing files. You may need to increase this # limit if you have huge mailboxes. #vsz\_limit = \$default\_vsz\_limit # Max. number of IMAP processes (connections) #process\_limit = 1024 } service pop3 { # Max. number of POP3 processes (connections) #process\_limit = 1024 } service submission { # Max. number of SMTP Submission processes (connections) #process\_limit = 1024 } service auth { # auth\_socket\_path points to this userdb socket by default. It's typically # used by dovecot-lda, doveadm, possibly imap process, etc. Users that have # full permissions to this socket are able to get a list of all usernames and # get the results of everyone's userdb lookups. # # The default 0666 mode allows anyone to connect to the socket, but the # userdb lookups will succeed only if the userdb returns an "uid" field that # matches the caller process's UID. Also if caller's uid or gid matches the # socket's uid or gid the lookup succeeds. Anything else causes a failure. # # To give the caller full permissions to lookup all users, set the mode to # something else than 0666 and Dovecot lets the kernel enforce the # permissions (e.g. 0777 allows everyone full permissions). unix\_listener auth-userdb { #mode = 0666 #user = #group = } # Postfix smtp-auth #unix\_listener /var/spool/postfix/private/auth { # mode = 0666 } # # Auth process is run as this user. #user = \$default\_internal\_user } service auth-worker { # Auth worker process is run as root by default, so that it can access # /etc/shadow. If this isn't necessary, the user should be changed to # \$default\_internal\_user. #user = root } service dict { # If dict proxy is used, mail processes should have access to its socket. # For example: mode=0660, group=vmail and global mail\_access\_groups=vmail unix\_listener dict { #mode = 0600 #user = #group = } }

-----



The section you're looking for that corresponds to the Postfix SMTP server integration with Dovecot for authentication is this part:

```
plaintext
```

```
# Postfix smtp-auth
#unix_listener /var/spool/postfix/private/auth {
#  mode = 0666
#}
```

This segment defines how Dovecot provides authentication for Postfix. The `unix_listener` specifies the socket path `/var/spool/postfix/private/auth`, which Postfix connects to for SASL (Simple Authentication and Security Layer) authentication.

If you're trying to set up authentication for Postfix via Dovecot, you'll need to uncomment (remove the leading `#` from) those lines. Here's how it should look:

```
plaintext
# Postfix smtp-auth
unix_listener /var/spool/postfix/private/auth {
  mode = 0666
}
```

Then you'll need to restart Dovecot (and possibly Postfix) for changes to take effect.

Remember, permissions set to 0666 mean that any user can read or write to the socket, but that's typically okay in this context since the socket resides in a directory (`/var/spool/postfix/private/`) with restricted permissions. However, always ensure your system's security based on your own requirements and situation.



What about: user = postfix group = postfix



Including `user = postfix` and `group = postfix` in the Postfix `unix_listener` configuration section within the Dovecot config specifies which user and group should own the socket file. This is a good practice for ensuring that the Postfix processes can connect to the Dovecot authentication socket without any permission issues.

If you decide to use this configuration, it would look like this:

```
plaintext
# Postfix smtp-auth
unix_listener /var/spool/postfix/private/auth {
  mode = 0660
  user = postfix
  group = postfix
}
```

Here's a quick breakdown:

- `mode = 0660`: This permission setting allows the owner (postfix user) and the group (postfix group) to read and write to the socket, but other users have no permissions. This is more restrictive and secure than 0666, which would allow any user on the system to read and write to the socket.
- `user = postfix` and `group = postfix`: These lines specify that the postfix user and group are the owner of the socket.

By using this configuration, you're adding an extra layer of security, ensuring that only Postfix can communicate with Dovecot for authentication purposes. After making these changes, you'll need to restart both Dovecot and Postfix to apply them.



How do I test postfix and dovecot



Testing Postfix and Dovecot involves verifying that each component of your mail server setup works as expected. Here are steps to test Postfix (SMTP) and Dovecot (IMAP/POP3):

## 1. Testing Postfix

### 1.1 Check if Postfix is running:

```
bash
sudo systemctl status postfix
```

### 1.2 Test local mail delivery:

Send a mail to the local user:

```
bash
echo "Test email body" | mail -s "Test Email" username@yourhostname
```

Then, check the mail for the user:

```
bash
cat /var/mail/username
```

or use a mail client like mutt:

```
bash
mutt
```

### 1.3 Test remote mail delivery:

Send an email to an external email address:

```
bash
echo "Test email body" | mail -s "Test Email" external.email@example.com
```

Check if the email is received at `external.email@example.com`.

## 2. Testing Dovecot

### 2.1 Check if Dovecot is running:

```
bash
sudo systemctl status dovecot
```

### 2.2 Test IMAP/POP3:

You can use the `openssl` command to test IMAP and POP3 connections:

#### IMAP:

```
bash
openssl s_client -connect localhost:993
```

#### POP3:

```
bash
openssl s_client -connect localhost:995
```

After connecting, for IMAP, you can use `. login username password` to attempt to log in.

For POP3, use `user username` and then `pass password`.

### 2.3 Use a mail client:

For a more real-world test, set up a mail client like Thunderbird, Outlook, or any other of your choosing. Configure an account with the IMAP/POP3 settings for Dovecot and SMTP settings for Postfix, then try to send and receive emails.

## 3. Check Logs

Always keep an eye on the logs for any issues or errors:

#### Postfix Logs:

```
bash
sudo tail -f /var/log/mail.log
```

Dovecot Logs:

Depending on your setup, Dovecot might also log to /var/log/mail.log, but it might have its own log files in /var/log/dovecot.log or similar.

4. External Tools

Several online tools can help you test your mail server's setup, such as:

- [MX Toolbox](#): This is a useful tool for checking your mail server's MX records, performing SMTP diagnostics, and checking for blacklists.
- [Mail Tester](#): This tool checks the quality of emails sent by your server and gives you a score based on various parameters like SPF, DKIM, and more.

Finally, remember to always check for returned emails or bounce-back messages after sending emails to external domains. They often provide clear reasons if there's an issue with your setup.

-----



```
tom@plan6:~$ openssl s_client -connect localhost:993 CONNECTED(00000003) Can't use SSL_get_servername depth=0 CN = localhost.localdomain
verify return:1 --- Certificate chain 0 s:CN = localhost.localdomain i:CN = localhost.localdomain --- Server certificate -----BEGIN CERTIFICATE-----
MIIC/TCCAeWgAwIBAgIUePjHlWMn4N3leX9R4f5l1HRisG0wDQYJKoZIhvcNAQEL
BQAwIDeEMBwGA1UEAwVbG9jYWxob3N0LmxvY2FsZG9tYWluMB4XDTEzMDkwMjIx
NTcxNloXDTEzMDgzMDIxNlNlcwIDAeMBwGA1UEAwVbG9jYWxob3N0LmxvY2Fs
ZG9tYWluMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtk00CaN+p7af
2ejHbo73jh12LK1BRV4COAO0x5e+RK62SYElebm+mZ5gZc8lw4XziLuPpdjgBkRx
eGUHdKf4bTuNwmYZ/LFvEVH60eQuDOhSl/K9zy613yhpFhuCSB/vWlmvawHTg9nz
rIBXJJMUgCwU61htYgYXinXg1QAVtcltop8pDn84Z1ykPpEdkl1kByYS4k3kn140
JUdyxR+iC8rTVp1p45ikb8Gwt+GWuXVJvHed2MmSVZhrT8f/WNdisTUIUi4zxeA
0HKx6MW4UD5qbKsyE2Jskwww3/IAOb9mNy10TGffKLr0QdpWlk6nwWO3FTnWyZE3
dvXguO+VbQIDAQABoy8wLTAJBgNVHRMEAjAAMCAGA1UdEQQZMBEjCFWxvY2FsG9Z
dC5sb2NhbGRvbWVpbiANBgkqhkiG9w0BAQsFAAOCAQEAtUs2zld3cKHHIk5dgFu
jkA1uNRGCBpFBFG6743GbClav5btIVUvcSURpJ2cqFaTcASjgxFzj0Do2kkXj2S
7bWfVNDSLV5yhFuZG/pLOvtPIPVtfk1ZVo03p4I1XKurE/cTIIzXA9edwibjJ6uf
EtMCFqSEjL1vSKkwPSwawwEaL5jFeCuDWnTBMUN8o5g8UoAN2SLuDR/z0t3jgPR
N9Qq1WCDcGHiiW1NbsUVd5LZ7M3stfOk9i/jbLuUpZt165v30RqEJpttJLs5XmPl
PGEIn2Cu65bqONpFhvUIEHdXbe8BAfG0Tg6omgvffAxpggwyZnurT/orRjtHgH 5Q== -----END CERTIFICATE----- subject=CN = localhost.localdomain
issuer=CN = localhost.localdomain --- No client certificate CA names sent Peer signing digest: SHA256 Peer signature type: RSA-PSS Server Temp Key:
X25519, 253 bits --- SSL handshake has read 1325 bytes and written 363 bytes Verification: OK --- New, TLSv1.3, Cipher is
TLS_AES_256_GCM_SHA384 Server public key is 2048 bit Secure Renegotiation IS NOT supported Compression: NONE Expansion: NONE No ALPN
negotiated Early data was not sent Verify return code: 0 (ok) --- Post-Handshake New Session Ticket arrived: SSL-Session: Protocol : TLSv1.3 Cipher :
TLS_AES_256_GCM_SHA384 Session-ID: C6D405DE397AD7055E32EF782DEC6EBF88F53D2D1D2270313875E9CBB6822FD4 Session-ID-ctx:
Resumption PSK: D73D6F1E6ABA1A86D633C979ACD9602F5AAE15C505D96240E5CA844F3BEB8E727EF676693485EBE505A503D0B37F8B54 PSK
identity: None PSK identity hint: None SRP username: None TLS session ticket lifetime hint: 7200 (seconds) TLS session ticket: 0000 - 6a 10 59 f5 2c 52
9f d6-af 6a e1 6c 60 6b 46 e6 j.Y.,R...j.l'kF. 0010 - 52 67 2f c6 d2 c2 2a 57-a9 db 1e a2 c2 ba 8c fc Rg/...*W..... 0020 - d3 e0 97 a4 6f 6d 30 f8-6e c7 48
42 8d d3 98 7d .....om0.n.HB...} 0030 - 8b a2 91 dd 4b 38 40 6a-23 b1 a2 e6 c1 80 43 0f ....K8@j#.....C. 0040 - 36 d6 0d 57 2c 81 95 37-6b c5 b7 f5 00 71
b9 f7 6..W...7k....q.. 0050 - 56 6c 93 2d b0 69 50 62-c9 2f 55 7a 46 96 1b a2 Vl.-iPb./UzF... 0060 - 9a f9 0b 48 c2 9a be 78-75 16 36 07 38 a6 da 61
...H...xu.6.8..a 0070 - 83 88 1e ca 52 ac ff a7-0b c3 20 15 84 d5 6c f4 ....R..... ..l. 0080 - 86 59 19 04 1f b4 0d 38-de 8e 12 6a 91 f3 06 f2 .Y.....8...j.... 0090
- 9c 81 b4 7b 5e f9 0b 38-4d 0e 64 c1 74 f1 35 ff ...{^..8M.d.t.5. 00a0 - 69 29 dd b6 d4 bd ac 60-ff 3c 5c f1 a4 2a d8 19 i).....^<..*. 00b0 - 37 c9 73 23 b6
19 7a 0d-24 6a 1f 57 ef f8 7d 89 7.s#..z.$j.W..}. Start Time: 1693851549 Timeout : 7200 (sec) Verify return code: 0 (ok) Extended master secret: no Max
Early Data: 0 --- read R BLOCK --- Post-Handshake New Session Ticket arrived: SSL-Session: Protocol : TLSv1.3 Cipher : TLS_AES_256_GCM_SHA384
Session-ID: BED0475A45AA4342EDC1E89602FB47830B84CD35940BC21E62E01BBEA3683FB9 Session-ID-ctx: Resumption PSK:
D42C54F982F7E1460DF3226E5EF1695327A20650BB9CBB59649A6C90EB75125967DF635570EEBF854FA899B0C9A0843E PSK identity: None PSK
identity hint: None SRP username: None TLS session ticket lifetime hint: 7200 (seconds) TLS session ticket: 0000 - 6a 10 59 f5 2c 52 9f d6-af 6a e1 6c 60
6b 46 e6 j.Y.,R...j.l'kF. 0010 - 50 61 06 53 fc 0b 18 09-fd f7 63 17 61 a4 91 2b Pa.S.....c.a.+ 0020 - f2 dc 98 b1 16 65 4f 87-4c 57 97 ae de d2 57 03
.....eO.LW.....W. 0030 - 11 7e 45 f5 19 26 75 9f-97 7d 1e 4e 45 56 80 27 .~E.&u..}.NEV.' 0040 - a7 25 99 bc 46 af ef e7-1e 59 59 60 46 ab 25 0c
.%...F...YY'F. 0050 - ae 5a 0e f0 46 7b 48 d1-46 55 4b d4 23 f4 73 63 .Z..F{H.FUK.#.sc 0060 - dd 44 c6 e3 c1 9e 7e 35-e4 2b fe 7f 85 ba 93 d7
.D....~5.+..... 0070 - cb cd 22 cd 6b d2 0f f9-d2 20 fc a9 cd 6d 5a a6 ..".k.....mZ. 0080 - 68 15 b0 65 40 13 f5 e2-f2 f5 5e c4 2d 69 c9 e2 h..e@.....^.-i..
0090 - a6 d0 81 6b 61 03 c6 6e-59 af 9b 4f 9d 46 40 25 ...ka..nY..O.F@% 00a0 - 36 ce a9 c5 c7 da cd 7b-75 a5 f8 37 71 73 27 04 6.....{u..7qs'. 00b0 - 1d
18 a0 c6 ff 78 e3 3b-cf 5b 69 4d 6a 18 6b 84 .....x.;[Mj.k. Start Time: 1693851549 Timeout : 7200 (sec) Verify return code: 0 (ok) Extended master secret:
no Max Early Data: 0 --- read R BLOCK * OK [CAPABILITY IMAP4rev1 SASL-IR LOGIN-REFERRALS ID ENABLE IDLE LITERAL+ AUTH=PLAIN
AUTH=LOGIN] Dovecot (Debian) ready.
```

-----



The output you provided is from `openssl s_client` and shows that you successfully connected to the IMAP server on port 993 using SSL/TLS encryption. Let's break down what you're seeing:

1. **Certificate Chain:** This section gives you the certificate details. In your case, the certificate's common name (CN) is `localhost.localdomain`, which suggests that you have a self-signed certificate (as the issuer is also `localhost.localdomain`).

2. **Server certificate:** This is the public certificate provided by the server. It is the one that the client will use to verify the server's identity and establish an encrypted connection.
3. **SSL handshake:** This section indicates the type and strength of the encryption, and if there were any issues establishing it. You have a successful handshake with TLSv1.3 using the TLS\_AES\_256\_GCM\_SHA384 cipher suite, which is a strong encryption method.
4. **Session Ticket:** This is related to the session resumption mechanism in TLS. The server can provide a session ticket that can be used by the client to resume a session without going through the full handshake process again.
5. The last line (\* OK [CAPABILITY . . .]) shows that Dovecot is working and ready to accept IMAP commands. The capabilities listed are the features supported by the server.

Now, to further test Dovecot (IMAP) functionality:

1. **Authenticate:** You can login to the server using the LOGIN command:

```
bash
a1 LOGIN username password
```

Replace username and password with the actual username and password you want to test.

2. **Check Mail:** After logging in, you can list the mailboxes:

```
bash
a2 LIST "" "*" "
```

And select a mailbox:

```
bash
a3 SELECT INBOX
```

3. **Logout:** To end the session, you can issue:

```
bash
a4 LOGOUT
```

In summary, you've successfully tested the SSL/TLS encryption of your IMAP server. To fully test the functionalities (like fetching emails, sending, deleting), you'll need to interact with it using IMAP commands.

**Note:** When testing like this, be cautious about entering sensitive data (like passwords) directly, especially if there might be any chance of others seeing or logging the input/output. If this is a production server or contains sensitive emails, using dedicated mail clients or testing tools in a safe environment is preferable.



You:

a1 LOGIN username password Would this a login / pw for a user of this debian system?



ChatGPT:

The line:

```
a1 LOGIN username password
```

is a command to log in to the IMAP server using the IMAP protocol. The a1 part is a client-provided identifier to match requests with responses (it could be anything, but in this case, it's a1). The LOGIN command is an IMAP command used to authenticate a user to the IMAP server.

The username and password are placeholders for the actual username and password of the email account on the IMAP server.

This does not necessarily mean that it's a login/password for a user of the Debian system in a general sense, but it's for an email account on the IMAP server (which happens to be running Dovecot on a Debian system). If the IMAP server (Dovecot in this case) is configured to use system authentication, then yes, the usernames and passwords could correspond to system users. But Dovecot (and other IMAP servers) can also be configured to use virtual users, database-backed users, or other authentication mechanisms that aren't directly tied to the system's user accounts.

In short, whether this is a login/password for a user of the Debian system depends on how Dovecot is configured on that system.

-----



You:

How can I check?



ChatGPT:

To determine how Dovecot is configured in terms of authentication, you'll need to look at the Dovecot configuration files. Here's how you can go about this:

1. **Locate the main configuration file:** Dovecot's main configuration file is typically located at `/etc/dovecot/dovecot.conf`. However, note that Dovecot's configuration can be split across multiple files in a directory, often `/etc/dovecot/conf.d/`.
2. **Check the authentication settings:** Open the main configuration file in a text editor, e.g.:

```
bash
sudo nano /etc/dovecot/dovecot.conf
```

Then, look for the `!include` directive, which includes other files. One of the files will typically handle authentication settings.

If there's a line like:

```
diff
!include auth-system.conf.ext
```

It suggests that system users are being used for authentication.

On the other hand, if you see something like:

```
diff
!include auth-sql.conf.ext
```

It means that authentication is being handled via an SQL database.

3. **Inspect the detailed authentication configuration:** Depending on the above step, navigate to the appropriate configuration file to get details about the authentication mechanism.
  - For system users: If using `auth-system.conf.ext`, it's typically pretty straightforward. System users will be authenticated against their system passwords.
  - For SQL users: If using `auth-sql.conf.ext`, the file will contain settings pointing to an SQL database. This database will contain the email users and their respective passwords.
  - There may also be other authentication mechanisms, like LDAP (`auth-ldap.conf.ext`). In such cases, you'll need to navigate to the respective configuration file and understand the setup.
4. **Password schemes:** Also, pay attention to the password schemes mentioned in the authentication configuration files. They determine how passwords are stored. For example, if you see `SCHEME=PLAIN`, passwords are stored in plain text. More secure schemes like `SCHEME=SHA256-CRYPT` denote that passwords are stored as hashed values.

Always remember to be cautious when handling configuration files, especially if they contain sensitive information. Don't inadvertently disclose or modify anything without understanding the implications.



You:

Where are the users defined? # Dovecot configuration file # If you're in a hurry, see <http://wiki2.dovecot.org/QuickConfiguration> # "doveconf -n" command gives a clean output of the changed settings. Use it # instead of copy&pasting files when posting to the Dovecot mailing list. # '#' character and everything after it is treated as comments. Extra spaces # and tabs are ignored. If you want to use either of these explicitly, put the # value inside quotes, eg.: key = "# char and trailing whitespace" # Most (but not all) settings can be overridden by different protocols and/or # source/destination IPs by placing the settings inside sections, for example: # protocol imap { }, local 127.0.0.1 { }, remote 10.0.0.0/8 { } # Default values are shown for each setting, it's not required to uncomment # those. These are exceptions to this though: No sections (e.g. namespace {}) # or plugin settings are added by default, they're listed only as examples. # Paths are also just examples with the real defaults being based on configure # options. The paths listed here are for configure --prefix=/usr # --sysconfdir=/etc --localstatedir=/var # Enable installed protocols !include\_try /usr/share/dovecot/protocols.d/\*.protocol # A comma separated list of IPs or hosts where to listen in for connections. # "\*" listens in all IPv4 interfaces, "::" listens in all IPv6 interfaces. # If you want to specify non-default ports or anything more complex, # edit conf.d/master.conf. #listen = \*, :: # Base directory where to store runtime data. #base\_dir = /var/run/dovecot/ # Name of this instance. In multi-instance setup doveadm and other commands # can use -i <instance\_name> to select which instance is used (an alternative # to -c <config\_path>). The instance name is also added to Dovecot processes # in ps output. #instance\_name = dovecot # Greeting message for clients. #login\_greeting = Dovecot ready. # Space separated list of trusted network ranges. Connections from these # IPs are allowed to override their IP addresses and ports (for logging and # for authentication checks). disable\_plaintext\_auth is also ignored for # these networks. Typically you'd specify your IMAP proxy servers here. #login\_trusted\_networks = # Space separated list of login access check sockets (e.g. tcpwrap) #login\_access\_sockets = # With proxy\_maybe=yes if proxy destination matches any of these IPs, don't do # proxying. This isn't necessary normally, but may be useful if the



destination # IP is e.g. a load balancer's IP. #auth\_proxy\_self = # Show more verbose process titles (in ps). Currently shows user name and # IP address. Useful for seeing who are actually using the IMAP processes # (eg. shared mailboxes or if same uid is used for multiple accounts). #verbose\_proctitle = no # Should all processes be killed when Dovecot master process shuts down. # Setting this to "no" means that Dovecot can be upgraded without # forcing existing client connections to close (although that could also be # a problem if the upgrade is e.g. because of a security fix). #shutdown\_clients = yes # If non-zero, run mail commands via this many connections to doveadm server, # instead of running them directly in the same process. #doveadm\_worker\_count = 0 # UNIX socket or host:port used for connecting to doveadm server #doveadm\_socket\_path = doveadm-server # Space separated list of environment variables that are preserved on Dovecot # startup and passed down to all of its child processes. You can also give # key=value pairs to always set specific settings. #import\_environment = TZ ## ## Dictionary server settings ## # Dictionary can be used to store key=value lists. This is used by several # plugins. The dictionary can be accessed either directly or through a # dictionary server. The following dict block maps dictionary names to URIs # when the server is used. These can then be referenced using URIs in format # "proxy::<name>". dict { #quota = mysql:/etc/dovecot/dovecot-dict-sql.conf.ext #expire = sqlite:/etc/dovecot/dovecot-dict-sql.conf.ext } # Most of the actual configuration gets included below. The filenames are # first sorted by their ASCII value and parsed in that order. The 00-prefixes # in filenames are intended to make it easier to understand the ordering. !include conf.d/\*.conf # A config file can also be tried to be included without giving an error if # it's not found: !include\_try local.conf



The provided configuration is the main `dovecot.conf` file. This file contains references to other configuration files and includes them for a more modular approach to configuration.

For user authentication, Dovecot usually relies on one of several methods: system users, database (SQL) users, LDAP users, etc. The authentication method and user details are usually stored in one of the `conf.d` files.

Here's what you can do to identify where users are defined:

1. **Check `conf.d` files:** Dovecot includes all configuration files from the `conf.d` directory, as indicated by the line `!include conf.d/*.conf` in the provided configuration. Among those files, one (or more) likely handles authentication settings.
2. **Look into the `auth*.conf.ext` files:** Navigate to `/etc/dovecot/conf.d/` and look for files that start with `auth`. For example:
  - `auth-system.conf.ext`: Uses system users for authentication.
  - `auth-sql.conf.ext`: Uses an SQL database for authentication.
  - `auth-ldap.conf.ext`: Uses LDAP for authentication.

You can open and inspect these files to determine which one is in use and where users are defined.

3. **For SQL-based authentication:** If using `auth-sql.conf.ext`, the file will have a reference to a database configuration file that contains connection details for the database. Within that database, there will be tables containing user information, including usernames and passwords.
4. **For LDAP-based authentication:** If using `auth-ldap.conf.ext`, the file will have settings related to your LDAP server, including where and how to look up users.
5. **For system-based authentication:** If `auth-system.conf.ext` is being used, then your Dovecot is authenticating against system users. In this case, the users are managed as regular system users on your Debian system (e.g., via `/etc/passwd` and `/etc/shadow`).

To really know where the users are defined, inspect the appropriate `auth*.conf.ext` file in the `/etc/dovecot/conf.d/` directory. If it's SQL-based, you'll then need to inspect the database; if it's system-based, you'd manage users as you would any system user on Debian.



You:

does this mean anything to you? `tom@plan6:/etc/dovecot/conf.d$ cat auth-master.conf.ext` # Authentication for master users. Included from 10-auth.conf. # By adding master=yes setting inside a passwd you make the passwd a list # of "master users", who can log in as anyone else. # <doc/wiki/Authentication.MasterUsers.txt> # Example master user passwd using passwd-file. You can use any passwd though. passwd { driver = passwd-file master = yes args = /etc/dovecot/master-users # Unless you're using PAM, you probably still want the destination user to # be looked up from passwd that it really exists. pass=yes does that. pass = yes } `tom@plan6:/e`



Yes, the provided configuration defines how "master users" authenticate in Dovecot. A master user is a special type of user that can log in as any other user without knowing their password. This can be useful for administrators who need to assist users or investigate issues without knowing users' actual passwords.

Here's a breakdown of the provided configuration:

1. **driver = passwd-file:** This line indicates that master user credentials are stored in a password file.
2. **master = yes:** This marks the `passwd` block as defining authentication for master users.
3. **args = /etc/dovecot/master-users:** This specifies the location of the password file containing the master users. Each entry in this file would typically be of the form `username:password`, where the password might be encrypted depending on the specific configuration.



4. **pass = yes:** This setting ensures that after a master user logs in as another user, Dovecot still checks if the impersonated user actually exists in the regular user database.

If you're investigating or managing the master users for this Dovecot installation, you would want to inspect (and possibly edit, with care) the `/etc/dovecot/master-users` file mentioned in the configuration. However, always be cautious when handling authentication configurations and sensitive files to avoid security risks.



`/etc/dovecot/master-users` is empty

-----



If the `/etc/dovecot/master-users` file is empty, it means there are currently no master users configured for Dovecot on your system. This could be intentional for security reasons, as having master users can introduce potential risks if not managed securely.

Here are a few things you might consider:

1. **Security:** It might be a good thing that there are no master users, especially if there's no real need for them. Master users can log into any user's mailbox, which is a powerful ability that can be a security concern if misused.
2. **Setting Up a Master User:** If you need a master user for administrative purposes, you can add one to the `master-users` file. Typically, the format would be `master_username:encrypted_password`. Ensure you use a strong, unique password, and consider using password hashing rather than plain text.

To generate a hashed password for Dovecot, you can use the `doveadm pw` command:

```
doveadm pw -s SHA512-CRYPT
```

You'll be prompted to enter a password, and then you'll receive a hashed version which you can place in the `master-users` file.

3. **Access Control:** Ensure that only trusted administrators can read or modify the `master-users` file. Typically, the file should have permissions such that only the root user or a designated group can read it.
4. **Documentation:** If you're managing or auditing a system, always check any available documentation or consult with colleagues who may have insight into why certain configurations are the way they are. It's possible there's a good reason the file is empty.
5. **Other Authentication Methods:** Remember, the configuration you showed is only one potential method for authenticating master users (using a `passwd-file`). Dovecot is flexible and can use other methods (like PAM, SQL, etc.), though the provided configuration specifically indicates the `passwd-file` method.

As always, when making changes related to security and authentication, proceed with caution and ensure you have backups and/or a way to revert changes in case something goes wrong.



Can I set it up so it emails can be sent or received through unix accounts on the system?

-----



Yes, Dovecot can be set up to use system accounts for email. Here's how to configure Dovecot and the associated Mail Transfer Agent (like Postfix) to use system accounts:

## 1. Dovecot Configuration:

For Dovecot to authenticate using system accounts, you'll generally want to use PAM (Pluggable Authentication Modules) or directly use the system's `passwd` database.

### 1. PAM Authentication:

In your Dovecot configuration, in `10-auth.conf`:

```
bash
```

```
passdb {
    driver = pam
}

userdb {
    driver = passwd
}
```

## 2. Mail Location:

In `10-mail.conf`, set the mail location. A common setting for system user mailboxes is:

```
bash
mail_location = mbox:~/mail:INBOX=/var/mail/%u
```

This uses the mbox format, which is traditional for UNIX systems. The user's mail is in `~/mail` and the inbox is in `/var/mail/username`.

## 2. Postfix Configuration:

If you're using Postfix as your MTA, you'll want to make sure it delivers mail to the location that Dovecot is expecting.

### 1. Local Delivery:

In `/etc/postfix/main.cf`, set:

```
bash
home_mailbox = mail/
```

This delivers mail to a `mail` directory in the user's home directory.

### 2. Dovecot as the Delivery Agent (optional, but often recommended):

If you want Dovecot to handle the final delivery (which can be beneficial if you're using features like Sieve), you'd set up Postfix to use Dovecot's LDA (Local Delivery Agent) or LMTP (Local Mail Transfer Protocol).

## 3. Secure the Setup:

- 1. Ensure Permissions are Correct:** The directories where mail is stored need to be accessible by the users. If you're using mbox, the `/var/mail` directory should be writable by the `mail` group, and each user's mailbox should be owned by them.
- 2. Use TLS:** Ensure that both Dovecot and Postfix are configured to use TLS for encrypted connections.
- 3. Firewall:** Only open the ports you need. Typically, for IMAP you'd open port 143 (or 993 for IMAPS), and for SMTP you'd open port 25 (and possibly 587 for submission).
- 4. Disable Unneeded Services:** If you're not using POP3, ensure it's disabled in Dovecot.

## 4. Test the Setup:

After setting everything up, it's critical to test:

- 1. Sending Email:** Use a client or `mail` command to send an email.
- 2. Receiving Email:** Send an email to a user on your system from an external source.
- 3. Fetching Email:** Configure an email client (like Thunderbird or Outlook) to connect to your Dovecot IMAP server and retrieve mail.

Remember to always backup your configuration files before making changes. Also, monitor logs (typically under `/var/log/`) for any errors or issues. Both Dovecot and Postfix are quite verbose and will generally give clear error messages.



how do I tell if I'm using a mail group?

-----



On a Unix or Linux system, you can check if you have a `mail` group by inspecting the `/etc/group` file or using the `getent` command.

Here's how you can do it:

- 1. Check `/etc/group`:**

You can use the `grep` command to search for a mail group:

```
bash
grep "^mail:" /etc/group
```

If you see a line starting with `mail:`, it means the mail group exists.

## 2. Use `getent`:

The `getent` command displays entries from databases supported by the Name Service Switch libraries, which is useful to get information in a consistent way, even if the backend is not the typical files (e.g., when using LDAP, NIS, etc.). To check for the mail group, run:

```
bash
getent group mail
```

If the mail group exists, this command will display its details.

If you're checking the mail group because you're configuring mail services, you might also need to ensure that specific directories or files are owned by or accessible to the mail group, such as the `/var/mail` directory. In that case, you can use the `ls -ld` command to check the ownership and permissions:

```
bash
ls -ld /var/mail
```

The output will show the owner and group associated with the directory, as well as its permissions.



```
tom@plan6:/etc/dovecot/conf.d$ grep "^mail:" /etc/group mail:x:8: tom@plan6:/etc/dovecot/conf.d$ getent group mail mail:x:8:
tom@plan6:/etc/dovecot/conf.d$
```



It looks like you have a mail group on your system with the group ID (GID) of 8. This group is typically used to provide the necessary permissions for mail-related processes.

If you're configuring Dovecot to deliver emails to system users' mailboxes, you might need to ensure that the mail directories and files are owned by or have the appropriate group permissions set for this mail group.

For Dovecot to work correctly with system user accounts:

1. Ensure the mail storage directories and files have the appropriate permissions. Typically, the mail storage for system users is in `/var/mail/USERNAME` (for mbox format) or somewhere in the user's home directory if you're using Maildir format.
2. Configure Dovecot to recognize and use system accounts. This is done using PAM (Pluggable Authentication Modules) or directly with the `passwd` file for authentication.

If you are aiming to use system accounts for email with Dovecot, you need to configure Dovecot's authentication mechanism to check against system users. This is typically done in Dovecot's `10-auth.conf` and `auth-system.conf.ext` files, ensuring that you're using the `pam` or `passwd-file` drivers, depending on your setup and preferences.

In the `10-auth.conf`, you should have something like:

```
diff
!include auth-system.conf.ext
```

And in the `auth-system.conf.ext`, it should use PAM (or another mechanism) to authenticate against system accounts:

```
passdb {
    driver = pam
}
userdb {
    driver = passwd
}
```

With this setup, when a user tries to authenticate, Dovecot will check against system users and their passwords.

Please review your Dovecot configuration to ensure that it's set up correctly for system account authentication and that mail storage has the right permissions. If you have more questions or encounter issues, don't hesitate to ask!



4. Test the Setup: After setting everything up, it's critical to test: Sending Email: Use a client or mail command to send an email. Receiving Email: Send an email to a user on your system from an external source. Fetching Email: Configure an email client (like Thunderbird or Outlook) to connect to your Dovecot IMAP server and retrieve mail



Absolutely. Testing is a crucial step after setting up any mail server component to ensure everything works as expected. Here's a simple guide for each step you've outlined:

### 1. Sending Email:

If you have an MTA like Postfix installed and configured on your server, you can use the `mail` command to send an email:

```
bash
echo "This is the body of the email" | mail -s "Test Subject" recipient@example.com
```

Replace `recipient@example.com` with a real email address where you can check for the receipt of the email.

### 2. Receiving Email:

To test receiving, you can send an email from an external source (like your personal email or another email service) to a user on your system. The email address would typically be `username@your-server-domain.com`, with `username` being a system user on the machine and `your-server-domain.com` being the domain or hostname of your server.

After sending the email, check the mailbox of the user on the server. If you're using the traditional mbox format, it's usually stored in `/var/mail/username`. If using Maildir, it might be in the user's home directory, e.g., `~username/Maildir/`.

You can use the `cat` command to check the contents of the mbox file or navigate the Maildir directory structure to check for new emails.

### 3. Fetching Email:

For this, you'd set up an email client to connect to your Dovecot IMAP server:

#### ◦ Thunderbird:

1. Go to Account Settings > Account Actions > Add Mail Account.
2. Enter the user's name, email address, and password.
3. Manual config: Set server hostname to your server's domain name or IP, port to 143 (or 993 for IMAPS), and select STARTTLS (or SSL/TLS for IMAPS).
4. Set the authentication method to Normal password.
5. Finish the setup and try fetching emails.

#### ◦ Outlook:

1. File > Add Account.
2. Choose Manual setup or additional server types > Next.
3. Select POP or IMAP.
4. Add user, email, account type as IMAP. Incoming mail server as your server domain or IP. Use the default port as 143 (or 993 for IMAPS).
5. Enter the user's password.
6. More Settings > Outgoing Server. Check My outgoing server (SMTP) requires authentication.
7. Go to the Advanced tab, set incoming server port. If using encryption, set it to STARTTLS (or SSL for IMAPS).
8. Finish and test fetching emails.

For all these tests, keep an eye on the logs (e.g., `/var/log/mail.log` or `/var/log/dovecot.log` depending on your setup and distribution). They can provide useful information if something doesn't work as expected.



Can I test the smtp server with authentication using something like this: `swaks --to twcookson@gmail.com --from billing@plan6.com --server plan6.com:587 --auth LOGIN --tls`

---



Yes, you can! `swaks` (which stands for "Swiss Army Knife for SMTP") is a versatile command-line utility for testing SMTP servers. The command you've provided is a great starting point for testing an SMTP server that requires authentication and uses TLS.

Here's a breakdown of the command:

- `--to twcookson@gmail.com`: This specifies the recipient's email address.
- `--from billing@plan6.com`: This specifies the sender's email address.
- `--server plan6.com:587`: This specifies the SMTP server and port number. Port 587 is typically used for submission (authenticated SMTP).
- `--auth LOGIN`: This specifies the authentication mechanism. LOGIN is a simple username/password mechanism. SMTP supports several authentication mechanisms, and LOGIN is one of the common ones.
- `--tls`: This tells `swaks` to use TLS when talking to the SMTP server.

You'll also likely need to specify the authentication credentials using `--auth-user` and `--auth-password`:

```
bash
swaks --to twcookson@gmail.com --from billing@plan6.com --server plan6.com:587 --auth LOGIN --tls --auth-user <USI
```

Replace `<USERNAME>` and `<PASSWORD>` with the actual SMTP authentication credentials.

Once you run the command, `swaks` will attempt to send an email to `twcookson@gmail.com` from `billing@plan6.com` using the SMTP server at `plan6.com` on port 587 with the specified authentication.

Pay attention to the output. `swaks` will provide verbose output, showing each step of the SMTP conversation. If there are any errors, they'll appear in this output, which can be very helpful for diagnosing issues.

-----



You:

Wow, I tested it from another computer to two email address and it worked flawlessly on both. Thank you! I've been working on this with you for 2 days and the dovecot postfix combo was the first to work

-----



That's fantastic news! Setting up mail servers, especially with authentication and other security features, can be a complex task. Dovecot and Postfix make a great combo, and when configured properly, they're both reliable and robust.

I'm really glad to have been of assistance, and it's great to hear that you've got everything up and running smoothly. Remember, always keep an eye on your mail server, regularly check logs, and consider setting up monitoring for any unusual activity. This will help ensure that your setup remains secure and operational.

If you have any more questions in the future, whether about email servers or anything else, don't hesitate to ask. Good luck with your Dovecot-Postfix setup, and take care!

-----



You:

So right now my password for the billing account is "raccoon" I don't think that's a great password, how do I change the account in unix to a more secure password, and does changing it in unix automatically update the email smtp credentials?

-----



Changing the password for a Unix user is straightforward. To change the password for the user `billing`, follow these steps:

#### 1. Change the password:

Open a terminal and type:

```
bash
sudo passwd billing
```

You will be prompted to enter a new password and then confirm it. Make sure to choose a strong password, preferably containing a mix of upper and lower case characters, numbers, and special symbols.

## 2. Impact on email SMTP credentials:

If Dovecot is configured to authenticate against system users (i.e., Unix users), then yes, changing the Unix password will also change the email authentication password. This is because Dovecot is looking at the same user credentials.

Similarly, if Postfix is set up to use the same Dovecot authentication for SMTP authentication (via SASL), then changing the Unix password will affect the SMTP credentials as well.

It's important to remember that after you change the password, you'll need to update the password in any email client software you're using to access the billing mailbox. Otherwise, you might face authentication errors the next time the client tries to connect.



You:

```
-----
This method will periodically be run by a cron in Django:
from accounts.models import CustomUser, Role
from lots.models import LotMetadata, CamMetadata
from ads.models import Ad
from django.utils import timezone

def generate_monthly_invoices():
    # Make invoices for advertisers
    # Change prices here to affect billing
    base_lot_price = 20
    per_camera_price = 20
    lot_operator_role = Role.objects.get(role_name="Lot Operator")
    lot_operators = CustomUser.objects.filter(role=lot_operator_role)

    for operator in lot_operators:
        print("Invoice for " + operator.first_name + " " + operator.last_name)
        print("Date " + timezone.localtime(timezone.now()).strftime('%Y-%m-%d %H:%M:%S'))
        print("Email: {operator.email}")

        lots = LotMetadata.objects.filter(owner=operator)
        print("\nCameras in service:")
        total_cameras = 0
        for lot in lots:
            print(' Lot id ' + lot.id + ':')
            cameras = CamMetadata.objects.filter(lot=lot)
            for camera in cameras:
                print(f' - Camera: {camera.name}')
            total_cameras = total_cameras + 1
        print(f'\n Total cameras {total_cameras} at ${per_camera_price} = ${per_camera_price * total_cameras}')
        final_invoice_price = base_lot_price + (total_cameras * per_camera_price)
        print(f'Invoice total:\nPer camera fee (${per_camera_price * total_cameras}) + base rate: (${base_lot_price}) = ${final_invoice_price}')

        # Make it so it creates new lot invoices based on this data:
        # Reference information: Models.py: class LotInvoice(models.Model):
        date_of_invoice = models.DateTimeField(auto_now_add=True)
        date_of_payment = models.DateTimeField(null=True, blank=True)
        invoice_id = models.AutoField(primary_key=True)
        customer = models.ForeignKey(CustomUser, on_delete=models.SET_NULL, null=True, blank=True)
        payment_method = models.ForeignKey('PaymentMethod', on_delete=models.SET_NULL, null=True, blank=True)
        has_been_paid = models.BooleanField(default=False)
        cameras = models.ManyToManyField(CamMetadata)
        payment_due = models.PositiveIntegerField(help_text="Amount due in pennies")
        is_monthly_invoice = models.BooleanField(default=True)
        description = models.TextField(max_length=3000, null=True, blank=True)

        def __str__(self):
            return str(self.invoice_id)

    # Serializers:
    from rest_framework import serializers
    from .models import LotInvoice, AdvertisementInvoice, PaymentMethod, CamMetadata, LotMetadata
    from accounts.serializers import UserSerializer

    class LotInvoiceSerializer(serializers.ModelSerializer):
        customer = UserSerializer(read_only=True)
        payment_method_name = serializers.CharField(source='payment_method.name', read_only=True)

        class Meta:
            model = LotInvoice
            fields = '__all__'

    class AdvertisementInvoiceSerializer(serializers.ModelSerializer):
        customer = UserSerializer(read_only=True)
        payment_method_name = serializers.CharField(source='payment_method.name', read_only=True)

        class Meta:
            model = AdvertisementInvoice
            fields = '__all__'

    class PaymentMethodSerializer(serializers.ModelSerializer):
        class Meta:
            model = PaymentMethod
            fields = '__all__'

    def to_representation(self, instance):
        rep = super().to_representation(instance)
        rep.pop('fake_credit_card_number', None)
        return rep

    class CreateLotInvoiceSerializer(serializers.ModelSerializer):
        cameras = serializers.PrimaryKeyRelatedField(many=True, queryset=CamMetadata.objects.all(), required=False, allow_empty=True)

        class Meta:
            model = LotInvoice
            fields = '__all__'

    class CreateAdvertisementInvoiceSerializer(serializers.ModelSerializer):
        lots_with_ads = serializers.PrimaryKeyRelatedField(many=True, queryset=LotMetadata.objects.all(), required=False, allow_empty=True)

        class Meta:
            model = AdvertisementInvoice
            fields = '__all__'
```



Alright, so based on the models and serializers you've provided, here's what you can do:

Firstly, you will need to update the `generate_monthly_invoices()` method to not just print invoices but to also create instances of the `LotInvoice` model and save them. Here's how you can modify the method to accomplish this:

```
python
from accounts.models import CustomUser, Role
from lots.models import LotMetadata, CamMetadata, LotInvoice
from django.utils import timezone

def generate_monthly_invoices():
    base_lot_price = 20
    per_camera_price = 20

    lot_operator_role = Role.objects.get(role_name="Lot Operator")
    lot_operators = CustomUser.objects.filter(role=lot_operator_role)

    for operator in lot_operators:
        # Building the invoice description string for later
        description_lines = [
            f"Invoice for {operator.first_name} {operator.last_name}",
            f"Date: {timezone.localtime(timezone.now()).strftime('%Y-%m-%d %H:%M:%S')}",
            f"Email: {operator.email}",
        ]
```

```

        "Cameras in service:"
    ]

    lots = LotMetadata.objects.filter(owner=operator)
    total_cameras = 0
    camera_list = [] # A list to hold the cameras for this operator

    for lot in lots:
        description_lines.append(f"Lot id {lot.id}:")
        cameras = CamMetadata.objects.filter(lot=lot)

        for camera in cameras:
            description_lines.append(f"Camera: {camera.name}")
            total_cameras += 1
            camera_list.append(camera)

    final_invoice_price = base_lot_price + (total_cameras * per_camera_price)
    description_lines.extend([
        f"Total cameras: {total_cameras} at ${per_camera_price} = ${per_camera_price * total_cameras}",
        f"Invoice total: ${final_invoice_price}"
    ])

    # Create a new LotInvoice instance
    invoice = LotInvoice(
        customer=operator,
        payment_due=final_invoice_price * 100, # Assuming this is in cents (pennies)
        description="\n".join(description_lines),
        is_monthly_invoice=True
    )
    invoice.save()

    # After saving the invoice, we can add the many-to-many relations
    invoice.cameras.set(camera_list)
    invoice.save()

    # Here you can add any additional operations, for example, sending this invoice by email to the user

```

A few things to note:

1. Instead of printing the invoice details, I'm accumulating them in the `description_lines` list, which is then joined into a string and saved as the description of the `LotInvoice`.
2. The `cameras` field is a many-to-many relationship. You have to save the `LotInvoice` instance first before you can set the related cameras. After creating and saving the instance, we set the cameras with the `invoice.cameras.set(camera_list)` method.
3. The `payment_due` field is mentioned to store the amount in pennies. So, the final price is multiplied by 100 to convert it into cents.
4. You can extend this function further, for example, by sending an email with the invoice details to the user. But this gives you the basic structure for creating the invoices based on the data you provided.
5. It's also a good practice to add error handling and logging for more robustness.



You:

`<td style={{textAlign:'left'}} colSpan="9"><strong>Description:</strong> {invoice.description}</td>` How do you make a newline or formatting in `invoice.description`?

-----



ChatGPT:

In the context you've provided, it looks like you're using JSX (commonly associated with React) to render an HTML table cell. If you want to display the newlines present in `invoice.description` as actual line breaks in the rendered output, you can use the CSS property `white-space: pre-line`.

Here's how you can apply it:

```

jsx
<td style={{textAlign:'left', whiteSpace: 'pre-line'}} colSpan="9">
  <strong>Description:</strong> {invoice.description}
</td>

```

The `white-space: pre-line` style will preserve newlines, whitespace, and sequences of whitespace in the `invoice.description` content. Newlines will be transformed into line breaks in the rendered HTML, allowing your description text to be properly formatted.

Make sure that the `invoice.description` string actually contains newline characters (like `\n`) at the points where you want to introduce line breaks. If you've followed the code from the previous answer, then the description string should have newline characters as intended.



Description: Invoice for Funky Chicken Date: 2023-09-04 Email: funky.chicken@example.com Cameras in service:      Lot id coldwater:      Camera: coldwatermi Total cameras: 1 at \$20 = \$20 Invoice total: \$40 How do I do what I'm trying to do here

-----



It looks like you're trying to format a string (in this case, a description for an invoice) with structured content, using whitespace and HTML entities for spacing and new lines. In a React JSX context, rendering HTML entities directly won't work as expected. Instead, you'll want to use JSX to generate the intended structure.

If you're aiming to generate a string like the one you've shown, and then render it in a JSX component, here's what you can do:

1. **Generate the formatted string:** Here's a hypothetical function based on your previous code, which returns the formatted description:

```
python
def generate_invoice_description(operator, lot, camera, total_cameras, per_camera_price):
    description = f"""
    Description:
    Invoice for {operator.first_name} {operator.last_name}

    Date: {timezone.localtime(timezone.now()).strftime('%Y-%m-%d')}

    Email: {operator.email}

    Cameras in service:
    \tLot id {lot.id}:

    \t\tCamera: {camera.name}

    Total cameras: {total_cameras} at ${per_camera_price} = ${total_cameras * per_camera_price}

    Invoice total: ${total_cameras * per_camera_price + base_lot_price}
    """
    return description
```

This function uses the `\t` character for tabs, which will be turned into spaces when rendered with the `white-space: pre-line` CSS rule.

2. **Render in JSX:**

Assuming you have a component where you want to render this description, you can use the following JSX:

```
jsx
<td style={{textAlign:'left', whiteSpace: 'pre-line'}} colSpan="9">
    {generate_invoice_description(operator, lot, camera, total_cameras, per_camera_price)}
</td>
```

Note: The `generate_invoice_description` function is a simplified representation; you'd likely need to modify it based on your exact use case and the structure of your data.



```
from accounts.models import CustomUser, Role from lots.models import LotMetadata, CamMetadata from ads.models import Ad from .models import
LotInvoice from django.utils import timezone def generate_monthly_invoices(): base_lot_price = 20 per_camera_price = 20 lot_operator_role =
Role.objects.get(role_name="Lot Operator") lot_operators = CustomUser.objects.filter(role=lot_operator_role) for operator in lot_operators: # Building the
invoice description string for later description_lines = [ f"Invoice for {operator.first_name} {operator.last_name}\n", f"Date:
{timezone.localtime(timezone.now()).strftime('%Y-%m-%d')}\n", f"Email: {operator.email}\n", "Cameras in service:" ] lots =
LotMetadata.objects.filter(owner=operator) total_cameras = 0 camera_list = [] # A list to hold the cameras for this operator for lot in lots:
description_lines.append(f"\tLot id {lot.id}:\n") cameras = CamMetadata.objects.filter(lot=lot) for camera in cameras: description_lines.append(f"\t\tCamera:
{camera.name}\n") total_cameras += 1 camera_list.append(camera) final_invoice_price = base_lot_price + (total_cameras * per_camera_price)
```



```

description_lines.extend([ f"Total cameras: {total_cameras} at ${per_camera_price} = ${per_camera_price * total_cameras}\n", f"Invoice total:
${final_invoice_price}" ]) # Create a new LotInvoice instance invoice = LotInvoice( customer=operator, payment_due=final_invoice_price * 100, #
Assuming this is in cents (pennies) description="\n".join(description_lines), is_monthly_invoice=True ) invoice.save() # After saving the invoice, we can add
the many-to-many relations invoice.cameras.set(camera_list) invoice.save() # Here you can add any additional operations, for example, sending this
invoice by email to the user # Make invoices for advertisers # Change prices here to affect billing base_ad_price = 10 per_ad_price = 10 advertiser_role =
Role.objects.get(role_name="Advertiser") advertisers = CustomUser.objects.filter(role=advertiser_role) for advertiser in advertisers: print("Invoice for " +
advertiser.first_name + " " + advertiser.last_name) print("Date " + timezone.localtime(timezone.now()).strftime("%Y-%m-%d %H:%M:%S")) print(f"Email:
{advertiser.email}") # Fetch all ads associated with this advertiser ads = Ad.objects.filter(user=advertiser) print("\nAds in service:") for ad in ads: print(f" -
Ad: {ad.name}") print(f"\n Total ad {len(ads)} at ${per_ad_price} = ${len(ads) * per_ad_price}") final_invoice_price = base_ad_price + (len(ads) *
per_ad_price) print(f"Invoice total is Per ad fee (${len(ads) * per_ad_price}) + base rate: (${base_ad_price}): ${final_invoice_price}") You made new
invoices for lot_invoices lot_operators correctly. Can you use similar logic to create AdvertisementInvoice for advertisers Here's models: from django.db
import models from accounts.models import CustomUser from lots.models import LotMetadata, CamMetadata class AdvertisementInvoice(models.Model):
date_of_invoice = models.DateTimeField(auto_now_add=True) date_of_payment = models.DateTimeField(null=True, blank=True) invoice_id =
models.AutoField(primary_key=True) customer = models.ForeignKey(CustomUser, on_delete=models.SET_NULL, null=True, blank=True)
payment_method = models.ForeignKey('PaymentMethod', on_delete=models.SET_NULL, null=True, blank=True) has_been_paid =
models.BooleanField(default=False) lots_with_ads = models.ManyToManyField(LotMetadata) payment_due =
models.PositiveIntegerField(help_text="Amount due in pennies") is_monthly_invoice = models.BooleanField(default=True) description =
models.TextField(max_length=3000, null=True, blank=True) # Optional field with max 3000 characters def __str__(self): return str(self.invoice_id) class
LotInvoice(models.Model): date_of_invoice = models.DateTimeField(auto_now_add=True) date_of_payment = models.DateTimeField(null=True,
blank=True) invoice_id = models.AutoField(primary_key=True) customer = models.ForeignKey(CustomUser, on_delete=models.SET_NULL, null=True,
blank=True) payment_method = models.ForeignKey('PaymentMethod', on_delete=models.SET_NULL, null=True, blank=True) has_been_paid =
models.BooleanField(default=False) cameras = models.ManyToManyField(CamMetadata) payment_due =
models.PositiveIntegerField(help_text="Amount due in pennies") is_monthly_invoice = models.BooleanField(default=True) description =
models.TextField(max_length=3000, null=True, blank=True) def __str__(self): return str(self.invoice_id) class PaymentMethod(models.Model):
CREDIT_CARD_CHOICES = ( ('VISA', 'Visa'), ('MASTER', 'MasterCard'), ('DISCOVER', 'Discover'), ) customer = models.ForeignKey(CustomUser,
on_delete=models.SET_NULL, null=True, blank=True) credit_card_type = models.CharField(max_length=10, choices=CREDIT_CARD_CHOICES)
fake_credit_card_number = models.CharField(max_length=16) # Assuming it's a fake number for placeholder, not for real transactions # Split
expiration_date into two fields expiration_month = models.PositiveIntegerField(choices=[(i, i) for i in range(1, 13)], help_text="Month in MM format",
default=13) # using 13 as an arbitrary value expiration_year = models.PositiveIntegerField(help_text="Year in YYYY format", default=0) # using 0 as an
arbitrary value name = models.CharField(max_length=255) billing_address = models.TextField() zip_code = models.CharField(max_length=6)
security_code = models.CharField(max_length=4) def __str__(self): return self.name Here's serializer: from rest_framework import serializers from
.models import LotInvoice, AdvertisementInvoice, PaymentMethod, CamMetadata, LotMetadata from accounts.serializers import UserSerializer class
LotInvoiceSerializer(serializers.ModelSerializer): customer = UserSerializer(read_only=True) payment_method_name =
serializers.CharField(source='payment_method.name', read_only=True) class Meta: model = LotInvoice fields = '__all__' class
AdvertisementInvoiceSerializer(serializers.ModelSerializer): customer = UserSerializer(read_only=True) payment_method_name =
serializers.CharField(source='payment_method.name', read_only=True) class Meta: model = AdvertisementInvoice fields = '__all__' class
PaymentMethodSerializer(serializers.ModelSerializer): class Meta: model = PaymentMethod fields = '__all__' def to_representation(self, instance): rep =
super().to_representation(instance) rep.pop('fake_credit_card_number', None) return rep class CreateLotInvoiceSerializer(serializers.ModelSerializer):
cameras = serializers.PrimaryKeyRelatedField( many=True, queryset=CamMetadata.objects.all(), required=False, allow_empty=True ) class Meta: model =
LotInvoice fields = '__all__' class CreateAdvertisementInvoiceSerializer(serializers.ModelSerializer): lots_with_ads = serializers.PrimaryKeyRelatedField(
many=True, queryset=LotMetadata.objects.all(), required=False, allow_empty=True ) class Meta: model = AdvertisementInvoice fields = '__all__'

```



Given the provided models and serializers, I'll help you extend the `generate_monthly_invoices` function to also generate `AdvertisementInvoice` instances for advertisers.

We'll create invoices for each advertiser with the following logic:

1. Fetch all the advertisers.
2. For each advertiser, retrieve all the ads associated with them.
3. Count the number of ads and calculate the final invoice price.
4. Build the description for the invoice.
5. Create a new `AdvertisementInvoice` instance for each advertiser with the calculated details.

Let's proceed with the code:

```

python
def generate_monthly_invoices():
    ...
    # The previous logic for lot_operators remains unchanged

    # Logic for generating AdvertisementInvoice for advertisers
    base_ad_price = 10
    per_ad_price = 10

    advertiser_role = Role.objects.get(role_name="Advertiser")
    advertisers = CustomUser.objects.filter(role=advertiser_role)

    for advertiser in advertisers:
        # Building the invoice description string for later
        description_lines = [
            f"Invoice for {advertiser.first_name} {advertiser.last_name}\n",
            f"Date: {timezone.localtime(timezone.now()).strftime('%Y-%m-%d')}\n",
            f"Email: {advertiser.email}\n\n",
            "Ads in service:\n"

```

```

]

ads = Ad.objects.filter(user=advertiser)
lot_list = [] # A list to hold the lots for this advertiser's ads

for ad in ads:
    description_lines.append(f"\tAd: {ad.name}\n")
    if ad.lot not in lot_list: # Assuming an Ad instance has a ForeignKey to LotMetadata
        lot_list.append(ad.lot)

final_invoice_price = base_ad_price + (len(ads) * per_ad_price)
description_lines.extend([
    f"\nTotal ads: {len(ads)} at ${per_ad_price} = ${len(ads) * per_ad_price}\n",
    f"Invoice total: ${final_invoice_price}\n"
])

# Create a new AdvertisementInvoice instance
ad_invoice = AdvertisementInvoice(
    customer=advertiser,
    payment_due=final_invoice_price * 100, # Assuming this is in cents (pennies)
    description="".join(description_lines),
    is_monthly_invoice=True
)
ad_invoice.save()

# After saving the ad_invoice, we can add the many-to-many relations
ad_invoice.lots_with_ads.set(lot_list)
ad_invoice.save()

# Here you can add any additional operations, for example, sending this ad_invoice by email to the user

```

With this logic added to your `generate_monthly_invoices` function, it will now generate `AdvertisementInvoice` instances for advertisers in addition to the `LotInvoice` instances for lot operators.

Note: Make sure to check and adjust your relationships between the models. For example, if an `Ad` instance doesn't have a direct `ForeignKey` to `LotMetadata`, you'll need to adjust the code accordingly.



You:

```

Fix this to be compatible with the many to many lots associated with each ad: # Change prices here to affect billing base_ad_price = 10 per_ad_price = 10
advertiser_role = Role.objects.get(role_name="Advertiser") advertisers = CustomUser.objects.filter(role=advertiser_role) for advertiser in advertisers: #
Building the invoice description string for later description_lines = [ f"Invoice for {advertiser.first_name} {advertiser.last_name}\n", f"Date:
{timezone.localtime(timezone.now()).strftime('%Y-%m-%d')}\n", f"Email: {advertiser.email}\n\n", "Ads in service:\n" ] ads = Ad.objects.filter(user=advertiser)
lot_list = [] # A list to hold the lots for this advertiser's ads for ad in ads: description_lines.append(f"\tAd: {ad.name}\n") if ad.lot not in lot_list: # Assuming an
Ad instance has a ForeignKey to LotMetadata lot_list.append(ad.lot) final_invoice_price = base_ad_price + (len(ads) * per_ad_price)
description_lines.extend([ f"\nTotal ads: {len(ads)} at ${per_ad_price} = ${len(ads) * per_ad_price}\n", f"Invoice total: ${final_invoice_price}\n" ]) # Create a
new AdvertisementInvoice instance ad_invoice = AdvertisementInvoice( customer=advertiser, payment_due=final_invoice_price * 100,
description="".join(description_lines), is_monthly_invoice=True ) ad_invoice.save() # After saving the ad_invoice, we can add the many-to-many relations
ad_invoice.lots_with_ads.set(lot_list) ad_invoice.save() Here is the ad model: class Ad(models.Model): advert_id = models.AutoField(primary_key=True)
name = models.CharField(max_length=255, verbose_name='Ad Name') start_date = models.DateField(verbose_name='Start Date', null=True,
blank=True) end_date = models.DateField(verbose_name='End Date', null=True, blank=True) user = models.ForeignKey(CustomUser,
on_delete=models.CASCADE, related_name='ads') url = models.URLField(max_length=1024, verbose_name='Target URL') lots =
models.ManyToManyField('lots.LotMetadata', blank=True, related_name='ads') # For counting impressions and clicks impressions =
models.PositiveIntegerField(default=0) clicks = models.PositiveIntegerField(default=0) # Top banner images (Vertical) top_banner_image1 =
models.ImageField(upload_to=ad_image_upload_path_top1, validators=[validate_file_size]) top_banner_image2 =
models.ImageField(upload_to=ad_image_upload_path_top2, validators=[validate_file_size]) top_banner_image3 =
models.ImageField(upload_to=ad_image_upload_path_top3, validators=[validate_file_size]) # Side banner images (Horizontal) side_banner_image1 =
models.ImageField(upload_to=ad_image_upload_path_side1, validators=[validate_file_size]) side_banner_image2 =
models.ImageField(upload_to=ad_image_upload_path_side2, validators=[validate_file_size]) side_banner_image3 =
models.ImageField(upload_to=ad_image_upload_path_side3, validators=[validate_file_size]) # Interval for image change in seconds
image_change_interval = models.PositiveIntegerField(default=10, help_text='Interval (in seconds) to switch between images') def __str__(self): return f"Ad
by {self.user.username}" def increment_impressions(self): self.impressions += 1 self.save() def increment_clicks(self): self.clicks += 1 self.save() Here is
the lots model: class LotMetadata(models.Model): id = models.CharField(max_length=100, primary_key=True) name =
models.CharField(max_length=255) owner = models.ForeignKey(CustomUser, on_delete=models.SET_NULL, null=True, blank=True) gps_coordinates =
models.CharField(max_length=255, null=True, blank=True) state = models.CharField(max_length=2, null=True, blank=True) zip =
models.CharField(max_length=5, null=True, blank=True) city = models.CharField(max_length=255, null=True, blank=True) def __str__(self): return
self.name class CamMetadata(models.Model): name = models.CharField(max_length=255, primary_key=True) lot = models.ForeignKey(LotMetadata,
on_delete=models.CASCADE) def __str__(self): return self.name

```

-----

You're wanting to generate AdvertisementInvoice instances based on the ads associated with each advertiser. The ads have a ManyToMany relationship with LotMetadata (lots), so we need to modify the code to account for this relationship when generating the invoices.

Let's use a set to track the unique lots associated with each ad for a given advertiser (since sets do not allow duplicate entries) and modify the invoice generation logic accordingly:

```
python
# Change prices here to affect billing
base_ad_price = 10
per_ad_price = 10

advertiser_role = Role.objects.get(role_name="Advertiser")
advertisers = CustomUser.objects.filter(role=advertiser_role)

for advertiser in advertisers:
    # Building the invoice description string for later
    description_lines = [
        f"Invoice for {advertiser.first_name} {advertiser.last_name}\n",
        f"Date: {timezone.localtime(timezone.now()).strftime('%Y-%m-%d')}\n",
        f"Email: {advertiser.email}\n\n",
        "Ads in service:\n"
    ]

    ads = Ad.objects.filter(user=advertiser)
    unique_lot_set = set() # A set to hold the unique lots for this advertiser's ads

    for ad in ads:
        description_lines.append(f"\tAd: {ad.name}\n")

        for lot in ad.lots.all(): # Iterating over each lot associated with the current ad
            unique_lot_set.add(lot)
            description_lines.append(f"\t\tLot: {lot.name}\n")

    final_invoice_price = base_ad_price + (len(ads) * per_ad_price)
    description_lines.extend([
        f"\nTotal ads: {len(ads)} at ${per_ad_price} = ${len(ads) * per_ad_price}\n",
        f"Invoice total: ${final_invoice_price}\n"
    ])

    # Create a new AdvertisementInvoice instance
    ad_invoice = AdvertisementInvoice(
        customer=advertiser,
        payment_due=final_invoice_price * 100,
        description="".join(description_lines),
        is_monthly_invoice=True
    )
    ad_invoice.save()

    # After saving the ad_invoice, we can add the many-to-many relations
    ad_invoice.lots_with_ads.set(unique_lot_set) # Using the set directly with set() function
    ad_invoice.save()
```

With this adjustment, we'll be correctly accounting for each lot associated with an advertiser's ads and adding it to the description. The set ensures that if multiple ads reference the same lot, the lot will only be counted once.



does unique\_lot\_set in any way effect the final price?

-----



No, in the provided code, the unique\_lot\_set does not directly affect the final price. The final price calculation is based solely on the number of ads associated with the advertiser:

```
python
final_invoice_price = base_ad_price + (len(ads) * per_ad_price)
```

The unique `lot_set` is used to gather all the unique lots associated with an advertiser's ads, to be linked later with the `AdvertisementInvoice` instance and added to the invoice description.

If you want the number of unique lots to affect the final price, you'd have to introduce that logic. But as of the provided code, it doesn't.



You:

```
I have a functioning and tested SMTP server I tested and I want to email billing descriptions to the email of the advertiser / lot_operator with bbc
twcookson@gmail.com. This command sends emails correctly over my server as I have configured it: swaks --to jedidork@gmail.com --from
billing@plan6.com --server plan6.com:25 --auth LOGIN --tls --auth-user billing --auth-password raccoonpw Make it so for each advertiser / lot_operator
receiving an invoice, they also get an email and I get a bcc too at twcookson@gmail.com: from accounts.models import CustomUser, Role from
lots.models import LotMetadata, CamMetadata from ads.models import Ad from .models import LotInvoice, AdvertisementInvoice from django.utils import
timezone def generate_monthly_invoices(): base_lot_price = 20 per_camera_price = 20 lot_operator_role = Role.objects.get(role_name="Lot Operator")
lot_operators = CustomUser.objects.filter(role=lot_operator_role) for operator in lot_operators: # Building the invoice description string for later
description_lines = [ f"Invoice for {operator.first_name} {operator.last_name}\n", f"Date: {timezone.localtime(timezone.now()).strftime('%Y-%m-%d')}\n",
f"Email: {operator.email}\n", "Cameras in service:" ] lots = LotMetadata.objects.filter(owner=operator) total_cameras = 0 camera_list = [] # A list to hold the
cameras for this operator for lot in lots: description_lines.append(f"\t\tLot id {lot.id}:\n") cameras = CamMetadata.objects.filter(lot=lot) for camera in cameras:
description_lines.append(f"\t\tCamera: {camera.name}\n") total_cameras += 1 camera_list.append(camera) final_invoice_price = base_lot_price +
(total_cameras * per_camera_price) description_lines.extend([ f"Total cameras: {total_cameras} at ${per_camera_price} = ${per_camera_price *
total_cameras}\n", f"Invoice total: ${final_invoice_price}" ]) # Create a new LotInvoice instance invoice = LotInvoice( customer=operator,
payment_due=final_invoice_price * 100, # Assuming this is in cents (pennies) description="".join(description_lines), is_monthly_invoice=True )
invoice.save() # After saving the invoice, we can add the many-to-many relations invoice.cameras.set(camera_list) invoice.save() # Here you can add any
additional operations, for example, sending this invoice by email to the user # Change prices here to affect billing base_ad_price = 10 per_ad_price = 10
advertiser_role = Role.objects.get(role_name="Advertiser") advertisers = CustomUser.objects.filter(role=advertiser_role) for advertiser in advertisers: #
Building the invoice description string for later description_lines = [ f"Invoice for {advertiser.first_name} {advertiser.last_name}\n", f"Date:
{timezone.localtime(timezone.now()).strftime('%Y-%m-%d')}\n", f"Email: {advertiser.email}\n\n", "Ads in service:\n" ] ads = Ad.objects.filter(user=advertiser)
unique_lot_set = set() # A set to hold the unique lots for this advertiser's ads for ad in ads: description_lines.append(f"\t\tAd: {ad.name}\n") for lot in
ad.lots.all(): # Iterating over each lot associated with the current ad unique_lot_set.add(lot) description_lines.append(f"\t\tLot: {lot.name}\n")
final_invoice_price = base_ad_price + (len(ads) * per_ad_price) description_lines.extend([ f"\nTotal ads: {len(ads)} at ${per_ad_price} = ${len(ads) *
per_ad_price}\n", f"Invoice total: ${final_invoice_price}" ]) # Create a new AdvertisementInvoice instance ad_invoice = AdvertisementInvoice(
customer=advertiser, payment_due=final_invoice_price * 100, description="".join(description_lines), is_monthly_invoice=True ) ad_invoice.save() # After
saving the ad_invoice, we can add the many-to-many relations ad_invoice.lots_with_ads.set(unique_lot_set) # Using the set directly with set() function
ad_invoice.save() user model: from django.db import models from django.contrib.auth.models import AbstractUser from django.core.validators import
MinLengthValidator class Role(models.Model): role_name = models.CharField(max_length=50) is_employee = models.BooleanField() def __str__(self):
return self.role_name class CustomUser(AbstractUser): role = models.ForeignKey(Role, on_delete=models.SET_NULL, null=True) email =
models.EmailField(unique=True) first_name = models.CharField(max_length=255, null=True, blank=True) last_name =
models.CharField(max_length=255, null=True, blank=True) company_name = models.CharField(max_length=255, null=True, blank=True)
company_address = models.CharField(max_length=255, null=True, blank=True) state = models.CharField(max_length=2, validators=
[MinLengthValidator(2)]), null=True, blank=True) city = models.CharField(max_length=255, null=True, blank=True) zip = models.CharField(max_length=5,
validators=[MinLengthValidator(5)]), null=True, blank=True) is_uninitialized = models.BooleanField(default=False) USERNAME_FIELD = 'email'
REQUIRED_FIELDS = ['username', 'first_name', 'last_name'] def __str__(self): return self.email from django.db import models from accounts.models
import CustomUser from lots.models import LotMetadata, CamMetadata class AdvertisementInvoice(models.Model): date_of_invoice =
models.DateTimeField(auto_now_add=True) date_of_payment = models.DateTimeField(null=True, blank=True) invoice_id =
models.AutoField(primary_key=True) customer = models.ForeignKey(CustomUser, on_delete=models.SET_NULL, null=True, blank=True)
payment_method = models.ForeignKey('PaymentMethod', on_delete=models.SET_NULL, null=True, blank=True) has_been_paid =
models.BooleanField(default=False) lots_with_ads = models.ManyToManyField(LotMetadata) payment_due =
models.PositiveIntegerField(help_text="Amount due in pennies") is_monthly_invoice = models.BooleanField(default=True) description =
models.TextField(max_length=3000, null=True, blank=True) # Optional field with max 3000 characters def __str__(self): return str(self.invoice_id) class
LotInvoice(models.Model): date_of_invoice = models.DateTimeField(auto_now_add=True) date_of_payment = models.DateTimeField(null=True,
blank=True) invoice_id = models.AutoField(primary_key=True) customer = models.ForeignKey(CustomUser, on_delete=models.SET_NULL, null=True,
blank=True) payment_method = models.ForeignKey('PaymentMethod', on_delete=models.SET_NULL, null=True, blank=True) has_been_paid =
models.BooleanField(default=False) cameras = models.ManyToManyField(CamMetadata) payment_due =
models.PositiveIntegerField(help_text="Amount due in pennies") is_monthly_invoice = models.BooleanField(default=True) description =
models.TextField(max_length=3000, null=True, blank=True) def __str__(self): return str(self.invoice_id) class PaymentMethod(models.Model):
CREDIT_CARD_CHOICES = ( ('VISA', 'Visa'), ('MASTER', 'MasterCard'), ('DISCOVER', 'Discover'), ) customer = models.ForeignKey(CustomUser,
on_delete=models.SET_NULL, null=True, blank=True) credit_card_type = models.CharField(max_length=10, choices=CREDIT_CARD_CHOICES)
fake_credit_card_number = models.CharField(max_length=16) # Assuming it's a fake number for placeholder, not for real transactions # Split
expiration_date into two fields expiration_month = models.PositiveIntegerField(choices=[(i, i) for i in range(1, 13)], help_text="Month in MM format",
default=13) # using 13 as an arbitrary value expiration_year = models.PositiveIntegerField(help_text="Year in YYYY format", default=0) # using 0 as an
arbitrary value name = models.CharField(max_length=255) billing_address = models.TextField() zip_code = models.CharField(max_length=6)
security_code = models.CharField(max_length=4) def __str__(self): return self.name
```



ChatGPT:

To send emails for each generated invoice, you'll first need to integrate your SMTP configurations with Django's email sending functions. However, as you have provided an SMTP command (swaks), we can call this command from within Django using Python's subprocess library.

Here's the modified `generate_monthly_invoices` function to send an email for each generated invoice:

1. First, you'll need to import the required modules:

```
python
```

```
import subprocess
from django.conf import settings
```

2. Then, add the email sending functionality in your `generate_monthly_invoices` function:

```
python
def send_invoice_email(recipient_email, subject, body):
    cmd = [
        'swaks',
        '--to', recipient_email,
        '--from', 'billing@plan6.com',
        '--server', 'plan6.com:25',
        '--auth', 'LOGIN',
        '--tls',
        '--auth-user', 'billing',
        '--auth-password', 'raccoonpw',
        '--header', f'Subject: {subject}',
        '--body', body,
        '--h-cc', 'twcookson@gmail.com' # Using CC here. If you prefer BCC, replace '--h-cc' with '--h-bcc'
    ]
    subprocess.run(cmd)

def generate_monthly_invoices():
    ...
    for operator in lot_operators:
        ...
        # Send Invoice Email
        email_subject = "Your Monthly Lot Invoice"
        send_invoice_email(operator.email, email_subject, "\n".join(description_lines))

    ...
    for advertiser in advertisers:
        ...
        # Send Invoice Email
        email_subject = "Your Monthly Advertisement Invoice"
        send_invoice_email(advertiser.email, email_subject, "\n".join(description_lines))
```

Please note:

- This approach uses `subprocess` to send emails, which will block the execution of your function until the email is sent. If you're sending a lot of emails, this could slow down the function considerably. For large scale, consider using asynchronous tasks with something like Celery.
- The email sending function has been separated into `send_invoice_email` for clarity. This function creates the command and calls `swaks` to send the email.
- Ensure that the `subprocess` library is safe to use in your environment and that there are no security concerns related to its use.
- Always use environment variables or secrets management for storing passwords and other sensitive details rather than hard-coding them in your code.



You:

how can I keep the password in a separate file which I can keep from being uploaded to git? from `accounts.models` import `CustomUser`, `Role` from `lots.models` import `LotMetadata`, `CamMetadata` from `ads.models` import `Ad` from `.models` import `LotInvoice`, `AdvertisementInvoice` from `django.utils` import `timezone` from `django.conf` import `settings` import `subprocess` def `send_invoice_email(recipient_email, subject, body)`: `cmd = [ 'swaks', '--to', recipient_email, '--from', 'billing@plan6.com', '--server', 'plan6.com:25', '--auth', 'LOGIN', '--tls', '--auth-user', 'billing', '--auth-password', 'raccoonpw', '--header', f'Subject: {subject}', '--body', body, '--h-cc', 'twcookson@gmail.com' # Using CC here. If you prefer BCC, replace '--h-cc' with '--h-bcc' ]` `subprocess.run(cmd)` def `generate_monthly_invoices()`: `base_lot_price = 20` `per_camera_price = 20` `lot_operator_role = Role.objects.get(role_name="Lot Operator")` `lot_operators = CustomUser.objects.filter(role=lot_operator_role)` for `operator` in `lot_operators`: # Building the invoice description string for later `description_lines = [ f'Invoice for {operator.first_name} {operator.last_name}\n', f'Date: {timezone.localtime(timezone.now()).strftime("%Y-%m-%d")}\n', f'Email: {operator.email}\n', "Cameras in service:" ]` `lots = LotMetadata.objects.filter(owner=operator)` `total_cameras = 0` `camera_list = []` # A list to hold the cameras for this operator for `lot` in `lots`: `description_lines.append(f'\tLot id {lot.id}:\n')` `cameras = CamMetadata.objects.filter(lot=lot)` for `camera` in `cameras`: `description_lines.append(f'\t\tCamera: {camera.name}\n')` `total_cameras += 1` `camera_list.append(camera)` `final_invoice_price = base_lot_price + (total_cameras * per_camera_price)` `description_lines.extend([ f'Total cameras: {total_cameras} at ${per_camera_price} = ${per_camera_price * total_cameras}\n', f'Invoice total: ${final_invoice_price}'])` # Create a new `LotInvoice` instance `invoice = LotInvoice( customer=operator, payment_due=final_invoice_price * 100, # Assuming this is in cents (pennies) description="\n".join(description_lines), is_monthly_invoice=True )` `invoice.save()` # After saving the invoice, we can add the many-to-many relations `invoice.cameras.set(camera_list)` `invoice.save()` `email_subject = "Your Monthly Lot Invoice"` `send_invoice_email(operator.email, email_subject, "\n".join(description_lines))` # Here you can add any additional operations, for example, sending this invoice by email to the user # Change prices here to affect billing `base_ad_price = 10` `per_ad_price = 10` `advertiser_role = Role.objects.get(role_name="Advertiser")` `advertisers = CustomUser.objects.filter(role=advertiser_role)` for `advertiser` in `advertisers`: # Building the invoice description string for later `description_lines = [ f'Invoice for {advertiser.first_name} {advertiser.last_name}\n', f'Date: {timezone.localtime(timezone.now()).strftime("%Y-%m-%d")}\n', f'Email: {advertiser.email}\n\n', "Ads in service:\n" ]` `ads = Ad.objects.filter(user=advertiser)` `unique_lot_set = set()` # A set to hold the unique lots for this advertiser's ads for `ad` in `ads`: `description_lines.append(f'\tAd: {ad.name}\n')` for `lot` in `ad.lots.all()`: # Iterating over each lot associated with the current ad `unique_lot_set.add(lot)` `description_lines.append(f'\t\tLot: {lot.name}\n')` `final_invoice_price = base_ad_price + (len(ads) * per_ad_price)` `description_lines.extend([ f'\nTotal ads: {len(ads)} at ${per_ad_price} = ${len(ads) *`

```
per_ad_price}\n", f"Invoice total: ${final_invoice_price}\n" ]) # Create a new AdvertisementInvoice instance ad_invoice = AdvertisementInvoice(
customer=advertiser, payment_due=final_invoice_price * 100, description="".join(description_lines), is_monthly_invoice=True ) ad_invoice.save() # After
saving the ad_invoice, we can add the many-to-many relations ad_invoice.lots_with_ads.set(unique_lot_set) # Using the set directly with set() function
ad_invoice.save() email_subject = "Your Monthly Advertisement Invoice" send_invoice_email(advertiser.email, email_subject, "".join(description_lines))
```



To keep your SMTP password (or any other sensitive data) outside of your code and prevent it from being uploaded to git, you can use a combination of environment variables and a `.gitignore` file. Here are the steps:

1. **Environment Variables:** Store your sensitive data in environment variables. This way, the actual values won't be in your source code, and they can be easily changed without modifying the code.
2. **.gitignore:** Use this file to specify which files and directories git should ignore. You can have configuration files that contain sensitive data and are read by your application but ignored by git.

### Steps:

1. Create a `.env` file at the root of your Django project. Add your sensitive data there:

```
makefile
SMTP_PASSWORD=raccoonpw
```

2. Add `.env` to your `.gitignore` file, so it won't be tracked by git:

```
bash
.env
```

3. Install `python-decouple` which will help you retrieve the environment variable in a way that also provides default values and conversion to correct types.

```
bash
pip install python-decouple
```

4. Update your code to use this:

```
python
from decouple import config

SMTP_PASSWORD = config('SMTP_PASSWORD')
```

5. Now in your `send_invoice_email` function, you can update the password section:

```
python
'--auth-password', SMTP_PASSWORD,
```

6. When deploying or running your app, ensure the environment variables are set correctly. You can use the `.env` file for local development. There are many tools and ways to set these for production. For instance, if you're using Docker, you'd pass them as environment variables in your `docker-compose.yml` or using `-e` with `docker run`. If you're deploying to a platform like Heroku, AWS, etc., they all provide ways to set environment variables for your app.
7. Periodically, it's a good idea to change sensitive data like passwords. Since you're using environment variables, changing them is straightforward without any code changes.

By following these steps, you ensure sensitive data is kept out of your version control system and can easily manage and change it when needed.



in a env how do I make requirements.txt



To generate a `requirements.txt` file from your virtual environment (venv), follow these steps:



### 1. Activate Your Virtual Environment:

If you are using `venv`, the activation commands are:

- **Linux/macOS:**

```
bash
source /path_to_venv/bin/activate
```

- **Windows:**

```
bash
\path_to_venv\Scripts\activate
```

### 2. Generate the `requirements.txt`:

Once you've activated your virtual environment, you can generate the `requirements.txt` by running:

```
bash
pip freeze > requirements.txt
```

This command lists all the installed packages in the active environment and writes them with their versions to the `requirements.txt` file.

### 3. Commit and Push to Version Control:

After generating the `requirements.txt`, you can commit it to your version control system:

```
bash
git add requirements.txt
git commit -m "Added requirements.txt"
git push
```

This ensures others collaborating on your project can easily replicate the environment by using the `requirements.txt` file.

When they clone or pull the repository, they can set up the same environment by creating a new virtual environment and then running:

```
bash
pip install -r requirements.txt
```

This will install all the packages listed in the `requirements.txt` file into the new virtual environment.

-----



You:

how do you install swaks

-----



ChatGPT:

`swaks` (Swiss Army Knife for SMTP) is a command-line tool for testing SMTP setups. You can install it using various methods depending on your operating system:

### 1. On Debian-based Linux (e.g., Ubuntu):

```
bash
sudo apt-get update
sudo apt-get install swaks
```

### 2. On Red Hat-based Linux (e.g., CentOS):

First, ensure you have the EPEL (Extra Packages for Enterprise Linux) repository enabled:

```
bash
sudo yum install epel-release
```

Then install `swaks`:

```
bash
sudo yum install swaks
```

### 3. On macOS:

If you're using [Homebrew](#):

```
bash
brew install swaks
```

### 4. From Source:

If none of the package managers have the latest version or if you want to install from source:

1. Download the latest version from the [official Swaks GitHub repository](#).
2. Then, follow the instructions provided in the README or INSTALL files.

### 5. On Other Systems:

If you are on a different system, check the package manager relevant to that OS. If swaks isn't available in the official repositories, consider installing it from source as described above.

Remember to check the swaks documentation or help page (`swaks --help`) after installation to get familiar with its usage.



I've got this `generate_monthly_invoices()` method in `tasks.py` in Django. I've got my Gunicorn server running Django all the time. I want to run `generate_monthly_invoices()` every month. I hear a cron is good for that. How do I send that up for `tasks.py` in the billing app on my Django app for `generate_monthly_invoices()`

-----



Yes, a cron job is perfect for periodically running tasks like this. To set up a cron job to run your `generate_monthly_invoices()` function from your Django app every month, you'll typically want to use Django's `manage.py` to ensure you're in the correct environment and have access to your models and settings. Here's a step-by-step guide:

#### 1. Create a Custom Management Command:

First, you'll want to create a custom Django management command to run `generate_monthly_invoices()`.

1.1. Inside your billing app directory (or the app where `tasks.py` is located), create a new directory called `management` and within that another directory called `commands`.

Your directory structure should look like:

```
lua
my_django_project/
|-- billing/
|   |-- management/
|   |   |-- commands/
|   |   |   |-- run_generate_monthly_invoices.py
```

1.2. Create a new file in `commands` named `run_generate_monthly_invoices.py`.

1.3. In `run_generate_monthly_invoices.py`, define the command:

```
python
from django.core.management.base import BaseCommand
from billing.tasks import generate_monthly_invoices

class Command(BaseCommand):
    help = 'Runs the generate_monthly_invoices method from tasks.py'

    def handle(self, *args, **kwargs):
        generate_monthly_invoices()
        self.stdout.write(self.style.SUCCESS('Successfully generated monthly invoices.'))
```



## 2. Set Up the Cron Job:

2.1. Open the crontab with the following command:

```
bash
crontab -e
```

2.2. Add an entry to run the command on the first day of every month (for instance). Here's what the cron line might look like:

```
javascript
0 0 1 * * /path/to/your/python /path/to/your/manage.py run_generate_monthly_invoices >> /path/to/logfile.log 2>&1
```

Breakdown:

- 0 0 1 \* \*: This specifies the schedule. It means "at minute 0, hour 0, day 1, of every month".
- /path/to/your/python: This should be the path to the Python interpreter that your Django project uses. You can find this path using the command `which python` or `which python3` (if you're using a virtual environment, activate it first).
- /path/to/your/manage.py: This is the path to your Django project's `manage.py` file.
- >> /path/to/logfile.log 2>&1: This is optional, but recommended. It logs the output and errors of the command to a file, so you can check if there were any issues.

Save and exit the editor to activate the cron job.

## 3. Test:

After setting everything up, you might want to adjust the cron timing temporarily to test if everything runs correctly, rather than waiting for the specified time.

Once you're sure the cron job is running correctly and your function works as expected, you can change the timing back to your original monthly schedule.

Note: Always remember to consider timezone differences, especially if your server's timezone is different from your application's timezone. Adjust cron timings accordingly.

-----



if I use env what would /path/to/your/python look like?

-----



If you're using a virtual environment (venv or virtualenv or similar) in your Django project, the Python interpreter you'd want the cron job to use would be the one inside that environment.

Here's how you can determine the path:

### 1. Activate your virtual environment:

If you're using venv:

```
bash
source /path/to/your/venv/bin/activate
```

Or if you're using virtualenv:

```
bash
source /path/to/your/env/bin/activate
```

Note: Replace /path/to/your/venv or /path/to/your/env with the actual path to your virtual environment directory.

### 2. Get the path for the Python interpreter:

While the virtual environment is active, run:

```
bash
which python
```

This will return the path to the Python interpreter being used in the activated environment. It will look something like:

```
bash
```

```
/path/to/your/venv/bin/python
```

or

```
bash  
/path/to/your/env/bin/python
```

### 3. Use this path in your cron job:

So, in the cron job you'd set up:

```
javascript  
0 0 1 * * /path/to/your/venv/bin/python /path/to/your/manage.py run_generate_monthly_invoices >> /path/to/log
```

Ensure you replace `/path/to/your/venv` with the correct path you obtained from the previous step.

This way, the cron job will run the command using the Python interpreter from your virtual environment, ensuring all your project dependencies are available.

-----