Sending Emails with Python

1. What Does Sending Email With Python Mean?

Python can connect to an email server, create an email message, and send that message to one or more recipients.

A typical flow looks like this:

Python Program
    ↓
Create Email
    ↓
Connect to SMTP Server
    ↓
Authenticate
    ↓
Send Message
    ↓
Recipient's Mail Server
    ↓
Recipient's Inbox

Python's standard library provides two important pieces:

- `email` — builds and represents email messages
- `smtplib` — communicates with SMTP servers and sends email

The `email` package itself does not send mail; `smtplib` handles communication with the SMTP server.

---

2. Why Automate Emails?

Email automation is useful when a program needs to notify someone automatically.

Examples:

- send a welcome email
- send a password reset message
- send an invoice
- send a daily report
- send an application notification
- send a backup status
- send an error alert
- send a generated PDF
- send a scheduled reminder
- notify an administrator when something fails

Instead of:

```
Human
 
Open email
 
Write message
 
Attach file
 
Send
```

you can build:

```
Python
 
Generate report
```

Receiving email involves other protocols and systems such as IMAP or POP3.

---

4. The Two Important Python Modules

email

Use `email` to construct the message.

```
from email.message import EmailMessage
```

It can represent:

- sender
- recipient
- subject
- body
- attachments
- HTML content
- multiple MIME parts

`EmailMessage` is the central modern class for constructing structured email messages.

---

smtplib

Use `smtplib` to connect to the SMTP server.

```
import smtplib
```

The general relationship is:

```
EmailMessage
    
Construct message
    
smtplib
    
Send message
```

---

5. Your First Email Message

Before sending anything, create the message.

```
from email.message import EmailMessage

message = EmailMessage()

message["Subject"] = "Hello from Python"
message["From"] = "sender@example.com"
message["To"] = "recipient@example.com"

message.set_content(
    "This email was sent using Python."
)
```

At this point, Python has created the email message.

It has **not** sent anything yet.

Think of it as:

```
EmailMessage
```

Subject
From
To
Body

---

6. Sending the Message

Now you need an SMTP server.

A simplified example:

```
import smtplib
from email.message import EmailMessage

message = EmailMessage()

message["Subject"] = "Hello from Python"
message["From"] = "sender@example.com"
message["To"] = "recipient@example.com"

message.set_content(
    "This email was sent using Python."
)

with smtplib.SMTP("smtp.example.com", 587) as smtp:
    smtp.starttls()
    smtp.login(
        "sender@example.com",
        "password"
    )
    smtp.send_message(message)
```

The official Python documentation recommends constructing messages with the `email` package and sending them using methods such as `send_message()`.

The SMTP hostname, port, and authentication method depend on your email provider.

---

7. Never Hard-Code Passwords

This is one of the most important rules.

Do not write:

```
password = "MyPassword123"
```

Your source code might eventually be:

- uploaded to GitHub
- shared with another developer
- stored in a backup
- included in logs
- accidentally exposed

Instead, use environment variables.

```
import os

email_password = os.environ["EMAIL_PASSWORD"]
```

Then:

```
smtp.login(
```

```
    sender_email,
    email_password
)
```

This separates:

```
Application Code
      +
Secret Configuration
```

---

8. Using Environment Variables

Suppose your environment contains:

```
EMAIL_ADDRESS=sender@example.com
EMAIL_PASSWORD=your-secret
```

Python:

```
import os

sender_email = os.environ["EMAIL_ADDRESS"]
password = os.environ["EMAIL_PASSWORD"]
```

Then:

```
message["From"] = sender_email

smtp.login(
    sender_email,
    password
)
```

For local development, a `.env` file can also be used with an appropriate environment-variable library, while the `.env` file should not be committed to Git.

---

9. Understanding SMTP Ports

Different SMTP configurations use different ports.

A common configuration is:

`587 → SMTP with STARTTLS`

Another common configuration is:

`465 → SMTP over SSL/TLS`

Python provides both:

`smtplib.SMTP(...)`

and:

`smtplib.SMTP_SSL(...)`

`SMTP_SSL` is used when SSL is required from the beginning of the connection; Python documents port 465 as the standard SMTP-over-SSL port.

Always follow the SMTP configuration provided by your email service.

---

10. STARTTLS

STARTTLS upgrades an existing SMTP connection to TLS encryption.

Example:

```
import smtplib

with smtplib.SMTP(
    "smtp.example.com",
    587
) as smtp:

    smtp.starttls()

    smtp.login(
        sender_email,
        password
    )
```

For stronger control over TLS configuration, Python supports passing an SSL context:

```
import ssl
import smtplib

context = ssl.create_default_context()

with smtplib.SMTP(
    "smtp.example.com",
    587
) as smtp:

    smtp.starttls(context=context)
```

```
    smtp.login(
        sender_email,
        password
    )
```

Python's documentation recommends `ssl.create_default_context()` for typical client-side TLS because it enables certificate validation and hostname checking.

---

11. SMTP Over SSL

Instead of STARTTLS, a provider may require SSL from the beginning.

```
import smtplib
import ssl

context = ssl.create_default_context()

with smtplib.SMTP_SSL(
    "smtp.example.com",
    465,
    context=context
) as smtp:

    smtp.login(
        sender_email,
        password
    )

    smtp.send_message(message)
```

The key difference is:

STARTTLS

Connect

SMTP

Upgrade to TLS

Authenticate

Send

versus:

SMTP_SSL

Secure connection

Authenticate

Send

---

12. Complete Basic Email Function

Instead of writing the same SMTP code everywhere, create a function.

```
import os
import ssl
import smtplib

from email.message import EmailMessage
```

```
def send_email(to, subject, body):
    sender = os.environ["EMAIL_ADDRESS"]
    password = os.environ["EMAIL_PASSWORD"]

    message = EmailMessage()

    message["From"] = sender
    message["To"] = to
    message["Subject"] = subject

    message.set_content(body)

    context = ssl.create_default_context()

    with smtplib.SMTP(
        "smtp.example.com",
        587
    ) as smtp:

        smtp.starttls(context=context)

        smtp.login(sender, password)

        smtp.send_message(message)
```

Now another part of your application can simply call:

```
send_email(
    "user@example.com",
    "Welcome",
    "Welcome to the application!"
)
```

This separates:

```
Email Service
      
Application Logic
```

---

13. Sending to Multiple Recipients

You can provide multiple recipients.

```
message["To"] = [
    "user1@example.com",
    "user2@example.com"
]
```

Or:

```
recipients = [
    "user1@example.com",
    "user2@example.com"
]

message["To"] = ", ".join(recipients)
```

Then:

```
smtp.send_message(message)
```

Python's email examples demonstrate sending a message to multiple recipients using a recipient list.

---

14. CC and BCC

Email messages can include:

```
To
CC
BCC
```

Example:

```
message["To"] = "user@example.com"

message["Cc"] = "manager@example.com"
```

BCC is slightly different because BCC recipients should not appear in the visible message headers.

For more controlled recipient handling, keep the envelope recipients separate from visible headers.

The important concept is:

```
Visible recipients
+
Envelope recipients
```

---

15. Plain Text Emails

A simple email can contain plain text.

```
message.set_content(
    """
```

```
Hello Hafsa,

Your report has been generated successfully.

Regards,
Python Automation
"""
)
```

Plain text is:

- simple
- accessible
- easy to generate
- easy to debug

For many automated notifications, plain text is completely sufficient.

---

16. HTML Emails

Emails can also contain HTML.

```
message.set_content(
    "Your report is ready."
)

message.add_alternative(
    """
    <html>
      <body>
        <h1>Report Ready</h1>
        <p>Your report has been generated.</p>
```

```
    </body>
  </html>
  """,
  subtype="html"
)
```

Now the email contains:

```
Plain text version
      +
HTML version
```

The recipient's email client can choose the appropriate representation.

---

17. Why Provide Both Plain Text and HTML?

A robust email can provide:

```
multipart/alternative
      
Plain text
      +
HTML
```

The HTML version can provide:

- headings
- paragraphs
- links
- tables
- basic formatting

The plain-text version remains useful when HTML is unavailable or undesirable.

---

18. Adding an Attachment

Suppose you have:

`report.pdf`

You can attach it.

```
from pathlib import Path

file_path = Path("report.pdf")

with file_path.open("rb") as file:
    message.add_attachment(
        file.read(),
        maintype="application",
        subtype="pdf",
        filename=file_path.name
    )
```

Then send the message normally.

Python's official email examples demonstrate adding binary files as MIME attachments using `EmailMessage.add_attachment()`.

---

19. Attaching Different File Types

For a CSV:

```
message.add_attachment(
    data,
    maintype="text",
    subtype="csv",
    filename="report.csv"
)
```

For a ZIP:

```
message.add_attachment(
    data,
    maintype="application",
    subtype="zip",
    filename="backup.zip"
)
```

For an image:

```
message.add_attachment(
    image_data,
    maintype="image",
    subtype="png",
    filename="chart.png"
)
```

The attachment's MIME type tells the email client what kind of content it is.

---

20. Automatically Detecting MIME Types

For a reusable email system, you can use Python's `mimetypes` module.

```
import mimetypes
```

```
content_type, encoding = mimetypes.guess_type(
    "report.pdf"
)

print(content_type)
```

For example:

```
application/pdf
```

Then split the type:

```
maintype, subtype = content_type.split("/", 1)
```

This becomes useful when your program accepts different file types.

Python's official email examples use MIME type detection when attaching files.

---

21. Reusable Attachment Function

```
from pathlib import Path
import mimetypes


def attach_file(message, file_path):
    path = Path(file_path)

    content_type, encoding = mimetypes.guess_type(path)

    if content_type is None or encoding is not None:
        content_type = "application/octet-stream"
```

```
    maintype, subtype = content_type.split("/", 1)

    with path.open("rb") as file:
        message.add_attachment(
            file.read(),
            maintype=maintype,
            subtype=subtype,
            filename=path.name
        )
```

Now:

```
attach_file(
    message,
    "report.pdf"
)
```

---

22. Sending Generated Reports

Email automation becomes particularly useful when combined with other Python skills.

Imagine:

```
Python
 
Generate CSV
 
Generate PDF
 
Create Email
 
```

Attach Files

Send

For example, an application might generate a daily report and email it automatically.

This combines:

- file handling
- data processing
- email
- environment variables
- logging
- scheduling

---

23. Email Automation With a Report

Conceptual workflow:

```
def generate_report():
    # Generate report
    pass


def send_report():
    # Create email
    # Attach report
    # Send email
    pass
```

```
report = generate_report()

send_report()
```

The important design principle is to keep:

```
Report Generation
```

separate from:

```
Email Delivery
```

That makes the system easier to test and maintain.

---

24. Sending Personalized Emails

Suppose you have:

```
users = [
   {
      "name": "Hafsa",
      "email": "hafsa@example.com"
   },
   {
      "name": "Asim",
      "email": "asim@example.com"
   }
]
```

You can personalize the body:

```
for user in users:

    body = f"""
Hello {user["name"]},

Your report is ready.

Regards,
The Team
"""

    send_email(
        user["email"],
        "Your Report",
        body
    )
```

This can be useful for legitimate application notifications and user-specific reports.

For bulk email, however, you must also consider provider limits, consent, unsubscribe requirements, deliverability, and anti-spam rules.

---

25. Email Templates

Instead of creating long strings everywhere, create templates.

```
def build_welcome_email(name):
    return f"""
Hello {name},

Welcome to the platform.

Your account is now ready.
```

```
Regards,
The Team
"""
```

Then:

```
body = build_welcome_email("Hafsa")
```

For larger systems, templates can live in separate files.

Example:

```
templates/
    welcome.txt
    welcome.html
    report.txt
    report.html
```

This separates email presentation from application logic.

---

26. Subject Lines Should Be Meaningful

Bad:

```
message["Subject"] = "Hi"
```

Better:

```
message["Subject"] = "Your monthly Python report"
```

For automated emails, a useful subject should help the recipient understand why they received the email.

Examples:

```
Daily Backup Completed
Your Account Has Been Created
Weekly Resource Report
Payment Confirmation
Automation Failed
```

---

27. Handling Email Errors

Email sending can fail.

Possible causes:

- incorrect credentials
- SMTP server unavailable
- network failure
- invalid recipient
- TLS configuration problem
- authentication failure
- provider restrictions
- connection timeout

Example:

```
import smtplib

try:
```

```
    send_email(
        "user@example.com",
        "Test",
        "Hello"
    )

except smtplib.SMTPAuthenticationError:
    print("Authentication failed.")

except smtplib.SMTPException as error:
    print("Email failed:", error)
```

`smtplib` provides specific SMTP-related exceptions, including authentication and TLS-related errors.

---

28. Timeouts

Network operations should not wait forever.

You can specify a timeout:

```
with smtplib.SMTP(
    "smtp.example.com",
    587,
    timeout=30
) as smtp:
    ...
```

This means the application has a defined limit for waiting on the connection.

Network code should generally have explicit timeout behavior.

---

29. Logging Email Operations

Do not rely only on:

```
print("Email sent")
```

Use logging for application-level automation.

```
import logging

logging.basicConfig(level=logging.INFO)

logging.info(
    "Sending report email to %s",
    recipient
)
```

After successful delivery:

```
logging.info(
    "Report email sent successfully"
)
```

On failure:

```
logging.exception(
    "Failed to send report email"
)
```

Avoid logging passwords, authentication tokens, or sensitive email content.

---

30. Building an Email Service

Instead of scattering email code throughout an application, create an email service.

```
class EmailService:

    def __init__(
        self,
        smtp_host,
        smtp_port,
        sender,
        password
    ):
        self.smtp_host = smtp_host
        self.smtp_port = smtp_port
        self.sender = sender
        self.password = password

    def send(self, recipient, subject, body):

        message = EmailMessage()

        message["From"] = self.sender
        message["To"] = recipient
        message["Subject"] = subject

        message.set_content(body)

        context = ssl.create_default_context()

        with smtplib.SMTP(
            self.smtp_host,
            self.smtp_port
```

```
        ) as smtp:

            smtp.starttls(context=context)

            smtp.login(
                self.sender,
                self.password
            )

            smtp.send_message(message)
```

Application code can then use:

```
email_service.send(
    recipient,
    "Welcome",
    "Welcome to the platform."
)
```

This follows the same separation-of-responsibilities principles used in maintainable Python applications.

---

31. Testing Email Code

You do not want every automated test to send a real email.

A test might accidentally send:

```
100 test emails
```

Instead, separate:

Message Creation

from:

Message Delivery

For example:

```
def build_message(
    sender,
    recipient,
    subject,
    body
):
    message = EmailMessage()

    message["From"] = sender
    message["To"] = recipient
    message["Subject"] = subject

    message.set_content(body)

    return message
```

Now you can test:

```
message = build_message(
    "sender@example.com",
    "user@example.com",
    "Welcome",
    "Welcome!"
)

assert message["Subject"] == "Welcome"
```

```
assert message["To"] == "user@example.com"
```

No email was actually sent.

---

32. Local Email Testing

For development, you can avoid sending real messages by using a local SMTP debugging server.

Python provides a useful development option:

```
python -m smtpd -c DebuggingServer -n localhost:1025
```

However, the `smtpd` module was deprecated and removed from newer Python versions, so modern projects should use a current local mail testing tool rather than relying on this old command.

The important idea is:

```
Development
  
Local/Test SMTP
  
No real recipient
```

This is safer than experimenting against real accounts.

---

33. Email Security

Email systems involve sensitive information.

Important rules:

Never expose credentials

Bad:

```
password = "secret"
```

Never commit secrets

Do not put:

```
EMAIL_PASSWORD=...
```

into a tracked `.env` file.

Use TLS

Use the secure SMTP configuration provided by your email service.

Validate recipients

Do not blindly send to arbitrary addresses supplied by untrusted input.

Avoid sensitive data in logs

Do not log:

```
password
token
full authentication headers
```

Limit access

Only the application components that need email credentials should have access to them.

---

34. Email Delivery Is Not the Same as Inbox Placement

Your Python program successfully calling:

`smtp.send_message(message)`

does not guarantee that the recipient sees the message in their inbox.

There are multiple stages:

`Python`
`SMTP Server`
`Recipient Mail Server`
`Spam / Filtering`
`Inbox or Spam Folder`

So distinguish:

`Message accepted by SMTP server`

from:

`Message successfully delivered to inbox`

This distinction becomes important when building production email systems.

---

35. Transactional Email vs Bulk Email

These are different use cases.

Transactional email

Triggered by an action.

Examples:

```
Account created
Password reset
Order confirmation
Report ready
```

Bulk email

One campaign or message sent to many recipients.

Examples:

```
Newsletter
Marketing campaign
Announcement
```

Bulk email requires additional considerations around:

- consent
- unsubscribe mechanisms
- provider limits
- reputation

- bounce handling
- spam regulations
- deliverability

A simple SMTP script is not automatically a complete bulk-email system.

---

36. When SMTP Is Enough

SMTP can be appropriate for:

- small internal tools
- personal automation
- application notifications
- low-volume reports
- development projects
- controlled transactional workflows

For larger production systems, dedicated email delivery services can provide additional infrastructure such as:

- delivery tracking
- bounce handling
- reputation management
- templates
- analytics
- API-based sending

The architectural concept remains:

```
Application
    
```

```
Email Provider
    
Recipient
```

---

37. Email Automation With Scheduling

You can combine email sending with scheduling.

For example:

```
Every day
  
Generate report
  
Send report
```

Conceptually:

```
def daily_job():
    report = generate_report()
    send_report(report)
```

The scheduler can invoke this function at the required time.

This connects email automation with the scheduling concepts you learned in Python automation.

---

38. A Complete Report Email Example

```
import os
```

```
import ssl
import smtplib
from pathlib import Path

from email.message import EmailMessage


def send_report(report_path, recipient):

    sender = os.environ["EMAIL_ADDRESS"]
    password = os.environ["EMAIL_PASSWORD"]

    report_path = Path(report_path)

    message = EmailMessage()

    message["From"] = sender
    message["To"] = recipient
    message["Subject"] = "Daily Report"

    message.set_content(
        """
Hello,

Your daily report is attached.

Regards,
Python Automation
"""
    )

    with report_path.open("rb") as file:
        message.add_attachment(
            file.read(),
```

```
            maintype="application",
            subtype="pdf",
            filename=report_path.name
        )

    context = ssl.create_default_context()

    with smtplib.SMTP(
        "smtp.example.com",
        587,
        timeout=30
    ) as smtp:

        smtp.starttls(context=context)

        smtp.login(
            sender,
            password
        )

        smtp.send_message(message)
```

The architecture is:

```
Report File
    
EmailMessage
    
Attachment
    
TLS SMTP Connection
    
Authentication
    
```

send_message()

---

39. A Better Architecture

For a real application:

```
app/
□
□□□ email_service.py
□□□ templates/
□   □□□ report.txt
□   □□□ report.html
□
□□□ reports/
□
□□□ config.py
□
□□□ main.py
```

Responsibilities:

email_service.py

Handles email creation and delivery.

templates/

Contains email content.

config.py

Loads configuration.

reports/

Contains generated files.

main.py

Coordinates the workflow.

```
main.py
  
Generate Report
  
Build Email
  
Attach Report
  
Email Service
  
SMTP
```

---

40. haas.dev Example

Imagine haas.dev generates a weekly resource report.

The system could do:

```
Every Friday
    
Collect resource statistics
    
Generate CSV
    
```

Generate summary

Create email

Attach CSV

Send report

The email could contain:

```
Weekly haas.dev Resource Report

New resources: 12
Updated resources: 5
Broken links: 1

The detailed CSV report is attached.
```

The important design is that email is only one component of the larger workflow.

Data

Processing

Report

Email

---

41. Mini Project: Automated Report Mailer

Build a command-line application that accepts:

`recipient`
`report file`

and sends the report.

Example:

```
python send_report.py user@example.com report.pdf
```

Your program should:

1. Read SMTP configuration from environment variables.
2. Validate that the report exists.
3. Build an `EmailMessage`.
4. Add subject and body.
5. Attach the report.
6. Connect securely to SMTP.
7. Authenticate.
8. Send the message.
9. Log success.
10. Handle failures.

Possible structure:

```
report_mailer/
□
□□□ send_report.py
□□□ email_service.py
□□□ config.py
□□□ requirements.txt
□□□ README.md
```

---

42. 5 Minute Challenge

Write a function:

```
send_email(
    recipient,
    subject,
    body
)
```

It should:

- read credentials from environment variables
- create an `EmailMessage`
- use TLS
- authenticate
- send the message
- handle SMTP errors

Then extend it to support:

```
send_email(
    recipient,
    subject,
    body,
    attachment="report.pdf"
)
```

---

43. Exercises

Exercise 1

Create a plain-text email.

Exercise 2

Send the email through an SMTP server.

Exercise 3

Move credentials into environment variables.

Exercise 4

Create a reusable `send_email()` function.

Exercise 5

Send an email to multiple recipients.

Exercise 6

Add CC support.

Exercise 7

Create an HTML email.

Exercise 8

Attach a PDF.

Exercise 9

Automatically detect an attachment's MIME type.

Exercise 10

Add logging and error handling.

Exercise 11

Build a daily report email system.

Exercise 12

Create an `EmailService` class.

Exercise 13

Write tests for message creation without sending real emails.

---

44. Mini Quiz

1. Which Python module communicates with SMTP servers?

A. `email`
B. `smtplib`
C. `pathlib`
D. `json`

Answer: B

2. Which class is commonly used to construct an email?

A. `EmailMessage`
B. `SMTPMessage`

C. `MailObject`
D. `EmailServer`

Answer: A

3. What does `starttls()` do?

A. Deletes the connection
B. Enables TLS encryption on the SMTP connection
C. Creates an email
D. Adds an attachment

Answer: B

4. Where should email credentials be stored?

A. Source code
B. Comments
C. Secure configuration such as environment variables or a secret manager
D. Git commit messages

Answer: C

5. Which method can send an `EmailMessage`?

A. `send_message()`
B. `send_file()`
C. `deliver()`
D. `push_email()`

Answer: A

---

45. Interview Questions

Beginner

1. What is SMTP?
2. What is `smtplib`?
3. What is the `email` package?
4. What is `EmailMessage`?
5. How do you set an email subject?
6. How do you specify the recipient?
7. How do you send a plain-text email?
8. What is the difference between SMTP and IMAP?

Intermediate

9. What is STARTTLS?
10. What is `SMTP_SSL`?
11. Why should credentials not be hard-coded?
12. How do you attach a file to an email?
13. What is MIME?
14. How do you send HTML email?
15. How do you send an email to multiple recipients?
16. How would you handle SMTP errors?
17. Why should email creation and email delivery be separated?
18. How would you test an email system without sending real emails?

Advanced

19. How would you design a reusable email service?
20. How would you implement email templates?
21. How would you handle retries?

22. How would you prevent duplicate emails?
23. How would you design a scheduled report-email system?
24. What is the difference between SMTP acceptance and inbox delivery?
25. What additional infrastructure is useful for large-scale transactional email?

---

46. Email Sending Cheat Sheet

Import

```
import smtplib

from email.message import EmailMessage
```

Create message

```
message = EmailMessage()
```

Subject

```
message["Subject"] = "Hello"
```

Sender

```
message["From"] = sender
```

Recipient

```
message["To"] = recipient
```

Body

```
message.set_content("Hello!")
```

HTML

```
message.add_alternative(
    "<h1>Hello!</h1>",
    subtype="html"
)
```

Attachment

```
message.add_attachment(
    data,
    maintype="application",
    subtype="pdf",
    filename="report.pdf"
)
```

TLS

```
smtp.starttls(context=context)
```

Login

```
smtp.login(sender, password)
```

Send

```
smtp.send_message(message)
```

SSL

```
with smtplib.SMTP_SSL(
    host,
    465,
    context=context
) as smtp:
    ...
```

---

47. The Email Automation Mental Model

Remember:

UNDERSTAND
Why does the application need to send email?
↓
DESIGN
What message, recipient, attachments and delivery method are needed?
↓
BUILD
Create EmailMessage
↓
CONNECT
Connect securely to SMTP
↓
AUTHENTICATE
Use secure credentials
↓
SEND
send_message()
↓
VERIFY
Handle errors and log the result
↓
MAINTAIN
Monitor delivery and provider limits

The goal is not:

> Make Python send an email.

The goal is:

> Build a reliable email-delivery component that another part of the application can safely use.

---

48. Key Takeaways

- Python can send email through SMTP.
- `email` constructs email messages.
- `smtplib` communicates with SMTP servers.
- `EmailMessage` is the modern high-level class for creating structured messages.
- Plain-text and HTML email can be combined.
- Files can be attached using `add_attachment()`.
- SMTP authentication should use securely managed credentials.
- TLS should be used according to the SMTP provider's configuration.
- `SMTP.starttls()` upgrades an SMTP connection to TLS.
- `SMTP_SSL` provides SSL/TLS from the beginning of the connection.
- Environment variables or secret managers are preferable to hard-coded credentials.
- Email creation should be separated from email delivery.
- Logging and error handling are important for automated email systems.
- Testing should avoid sending real emails whenever possible.
- SMTP acceptance does not guarantee inbox placement.
- Large-scale email delivery involves additional concerns such as consent, provider limits, bounces, and deliverability.
- Email becomes particularly powerful when combined with reports, file generation, automation, and scheduling.

Website:
https://dev-roast-app.vercel.app

More resources:
https://dev-roast-app.vercel.app/resources

Visit haas.dev for more resources and guides.