

Inicio (<https://recursospython.com/>)

Códigos de fuente (<https://www.recursospython.com/category/codigos-de-fuente/>)

Guías y manuales (<https://www.recursospython.com/category/guias-y-manuales/>)


Foro (<https://foro.recursospython.com/>)      Micro (<https://micro.recursospython.com/>)

Tutorial (<https://tutorial.recursospython.com/>)      Newsletter (<https://recursospython.com/newsletter/>)

Consultoría (<https://recursospython.com/consultoria/>)

Contacto (<https://recursospython.com/contacto/>)      Donar  (<https://recursospython.com/donar/>)

# Desarrollando una API REST con Twisted Klein

agosto 2, 2018 (<https://recursospython.com/guias-y-manuales/api-rest-twisted-klein/>) by [Recursos Python](https://recursospython.com/author/admin/) (<https://recursospython.com/author/admin/>)  (<https://recursospython.com/guias-y-manuales/api-rest-twisted-klein/#comments>) [Dejar un comentario](https://recursospython.com/guias-y-manuales/api-rest-twisted-klein/#respond) (<https://recursospython.com/guias-y-manuales/api-rest-twisted-klein/#respond>)

**Descarga:** [emailservice.zip](https://www.recursospython.com/wp-content/uploads/2018/08/emailservice.zip) (<https://www.recursospython.com/wp-content/uploads/2018/08/emailservice.zip>).

Klein es un pequeñísimo *web framework* montado sobre Twisted, la plataforma de red asincrónica de la que ya [hemos hablado bastante](https://www.recursospython.com/tag/twisted/) (<https://www.recursospython.com/tag/twisted/>), y Werkzeug (<http://werkzeug.pocoo.org/>), una librería para el desarrollo de aplicaciones WSGI. Puesto que sobre esta última se ha desarrollado el *microframework* Flask (<http://flask.pocoo.org/>), desarrollar aplicaciones en Klein resultará bastante familiar para aquellos que tengan alguna experiencia con él.

Por tratarse de un framework asincrónico, resulta ideal para desarrollar servicios web por cuanto puede atender a múltiples peticiones concurrentemente, a diferencia de los frameworks WSGI —como Django, web2py, Pyramid—.



## Últimas entradas

[Reproducir inyección](#)

[SQL en sqlite3 y](#)

[PyMySQL](#)

[\(https://recursospython.com/guias-y-manuales/reproducir-inyeccion-sql-en-sqlite3-y-pymysql/\)](https://recursospython.com/guias-y-manuales/reproducir-inyeccion-sql-en-sqlite3-y-pymysql/)

[Bloc de notas simple con](#)

[Tk \(tkinter\)](#)

[\(https://recursospython.com/codigos-de-fuente/bloc-de-notas-simple-con-tkinter/\)](https://recursospython.com/codigos-de-fuente/bloc-de-notas-simple-con-tkinter/)

[Examinar archivo o](#)

[carpeta en Tk \(tkinter\)](#)

[\(https://recursospython.com/guias-y-manuales/examinar-archivo-o-carpeta-en-tk-tkinter/\)](https://recursospython.com/guias-y-manuales/examinar-archivo-o-carpeta-en-tk-tkinter/)

En este artículo estaremos desarrollando una pequeña API REST ([https://es.wikipedia.org/wiki/Transferencia\\_de\\_Estado\\_Representacio](https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacio)) que permitirá enviar correos electrónicos vía SMTP y luego consultar su estado. Esto mejoraría, por ejemplo, la experiencia del usuario en una aplicación web que requiera el envío de correos, ya que no se tendrá que interactuar directamente con el servidor SMTP sino vía HTTP con nuestra API, que responderá considerablemente más rápido.

## Instalación

Antes de empezar, instalemos Klein vía `pip` (<https://www.recursospython.com/guias-y-manuales/instalacion-y-utilizacion-de-pip-en-windows-linux-y-os-x/>):

```
pip install klein
```

Este comando instalará todas las dependencias, incluyendo Twisted y Werkzeug.

## Primeros pasos

Bien, comenzaremos por definir la estructura básica de una aplicación de Klein, que correrá en la dirección local y en el puerto 7001.

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.
4.  from klein import Klein
5.
6.
7.  class EmailService:
8.      app = Klein()
9.
10.
11.  if __name__ == '__main__':
12.      emailservice = EmailService()
13.      emailservice.app.run('localhost', 7001)
```

A continuación expondremos la ruta `/email` definiendo el método `EmailService.email()`, que aceptará dos operaciones: `POST`, para enviar un correo, y `GET`, que retornará una lista con los mensajes enviados.

```
1.  @app.route('/email', methods=['GET', 'POST'])
2.  def email(self, request):
3.      if request.method == b'POST':
4.          return 'Enviar email.'
5.      elif request.method == b'GET':
6.          return 'Retornar lista de emails.'
```

[Múltiples configuraciones \(desarrollo/producción\) en Django \(https://recursospython.com/guias-y-manuales/multiples-configuraciones-desarrollo-produccion-en-django/\)](#)

[Buscar el archivo de mayor tamaño en una ruta \(https://recursospython.com/codigo-de-fuente/buscar-el-archivo-de-mayor-tamano-en-una-ruta/\)](#)

## Comentarios recientes

Recursos Python en [Generar código QR \(https://recursospython.com/guias-y-manuales/generar-codigo-qr/#comment-2586\)](#)

Joaquín en [Generar código QR \(https://recursospython.com/guias-y-manuales/generar-codigo-qr/#comment-2584\)](#)

Recursos Python en [pickle – Serialización de objetos \(https://recursospython.com/guias-y-manuales/pickle-serializacion-de-objetos/#comment-2435\)](#)

Recursos Python en [Lista desplegable \(Combobox\)](#)



Como observamos, el segundo argumento de todo método es siempre `request`, que contiene información sobre la petición HTTP. El valor de retorno de la función será enviado al usuario como respuesta.

Ejecutemos nuestro pequeño código y hagamos algunas pruebas usando la librería [Requests](http://docs.python-requests.org/en/master/) (<http://docs.python-requests.org/en/master/>) (si no la tienes instalada, simplemente ejecuta `pip install requests`).

```
1. >>> import requests
2. >>> url = 'http://localhost:7001/email'
3. >>> r = requests.get(url)
4. >>> r.text
5. 'Retornar lista de emails.'
6. >>> r = requests.post(url)
7. >>> r.text
8. 'Enviar email.'
```

Perfecto, nuestra aplicación responde correctamente a las peticiones. Primero desarrollaremos la operación `POST`. Haremos que se requieran los parámetros `to`, `subject` y `message`, que indicarán el recipiente del correo electrónico, el asunto y el mensaje.

Podemos acceder a los parámetros de la petición vía el diccionario `request.args`.

```
1. if request.method == b'POST':
2.     print(request.args[b'to'])
3.     print(request.args[b'subject'])
4.     print(request.args[b'message'])
```

Si hacemos la siguiente prueba:

```
1. >>> params = {'to': 'nombre@ejemplo.com', 'subject':
2. 'Test',
3. ...         'message': '¡Hola, mundo!'}
4. >>> requests.post(url, data=params)
```

Veremos en la consola cómo se imprimen los valores de los parámetros.

```
2018-07-22 15:02:48-0300 [-] [b'nombre@ejemplo.com']
2018-07-22 15:02:48-0300 [-] [b'Test']
2018-07-22 15:02:48-0300 [-] [b'\xc2\xa1Hola, mundo!']
```

Tendremos en cuenta que los parámetros son siempre listas, por lo que habremos de acceder a su primer elemento, que es una instancia del tipo `bytes` (nótese la `b` antes de las cadenas).

[en Tcl/Tk \(tkinter\)](https://recursospython.com/gui-y-manuales/lista-desplegable-combobox-en-tkinter/#comment-2434)  
(<https://recursospython.com/gui-y-manuales/lista-desplegable-combobox-en-tkinter/#comment-2434>)  
Herná en [Lista desplegable \(Combobox\)](https://recursospython.com/gui-y-manuales/lista-desplegable-combobox-en-tkinter/#comment-2424)  
[en Tcl/Tk \(tkinter\)](https://recursospython.com/gui-y-manuales/lista-desplegable-combobox-en-tkinter/#comment-2424)  
(<https://recursospython.com/gui-y-manuales/lista-desplegable-combobox-en-tkinter/#comment-2424>)

Pero antes me interesa chequear que todos los parámetros estén presentes.

```
1.         if request.method == b'POST':
2.             # Look for missing params.
3.             for param in (b'to', b'message', b'subject'):
4.                 if param not in request.args:
5.                     return "Missing parameter:
6.             '{}'.format(
                    param.decode('utf-8'))
```

```
1. >>> r = requests.post(url)
2. >>> r.text
3. "Missing parameter: 'to'."
```

Ahora bien, por lo general las API REST trabajan con los formatos JSON y XML en sus respuestas, no con texto plano ni código HTML. Optaremos por JSON, que es la más utilizada. Entonces una respuesta de nuestra aplicación se verá más o menos así:

```
{'succeed': true, 'status': 200, ...}
```

Es decir, siempre tendrá los valores `succeed` y `status`, que indicarán si la operación se ejecutó correctamente y el código HTTP de retorno (`200` indica una consulta exitosa). El resto de los valores retornados dependerá de la operación.

Pues bien, creemos un método que se ocupe de construir este valor de retorno como JSON y establezca la cabecera HTTP apropiada.

```
1.     def response(self, request, succeed=True, status=200,
2.     **kwargs):
3.         """
4.         Build the response body as JSON and set the proper
5.         content-type
6.         header.
7.         """
8.         request.setHeader('Content-Type',
9.         'application/json')
10.        request.setResponseCode(status)
11.        return json.dumps(
12.        {'succeed': succeed, 'status': status,
13.        **kwargs})
```

Recordemos importar el módulo json

(<https://www.recursospython.com/guias-y-manuales/json/>) al comienzo del archivo.

```
1. import json
```

Y por último modifiquemos el valor de retorno en caso que esté faltando algún parámetro a la operación `POST`.



```

1.         if param not in request.args:
2.             return self.response(
3.                 request,
4.                 succeed=False,
5.                 status=400,
6.                 reason="Missing parameter:
'{}'.format(
7.                     param.decode('utf-8'))
8.             )

```

```

1. >>> r = requests.post(url)
2. >>> r.json()
3. {'succeed': False, 'status': 400, 'reason': "Missing
parameter: 'to'."}

```

En este caso estamos retornando un código de error **400**, que indica una petición mal formada por el cliente. Puedes ver una lista de los códigos HTTP en [este enlace](https://es.wikipedia.org/wiki/Anexo:C%C3%B3digos_de_estado_HTTP) ([https://es.wikipedia.org/wiki/Anexo:C%C3%B3digos\\_de\\_estado\\_HTTP](https://es.wikipedia.org/wiki/Anexo:C%C3%B3digos_de_estado_HTTP))

## Manejando errores

Hay dos casos en los que Klein retornará dos errores HTTP automáticamente. Uno es cuando se intenta acceder a una ruta que no existe (**404**); otro, cuando se ejecuta una operación no definida sobre una ruta existente (**405**), por ejemplo, **PUT** o **DELETE** en `/email`. Veremos que la respuesta por defecto es un código HTML.

```

1. >>> r = requests.put(url)
2. >>> r
3. <Response [405]>
4. >>> r.text
5. '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2
Final//EN">\n<title>405 Method Not
6. Allowed</title>\n<h1>Method Not Allowed</h1>\n<p>The method
is not allowed for t
7. he requested URL.</p>\n'

```

Nuestra API debería ser capaz de manejar estos errores y retornar, en su lugar, una respuesta JSON acorde al formato que especificamos. Para ello vamos a importar las excepciones correspondientes.

```

1. from werkzeug.exceptions import NotFound, MethodNotAllowed

```

Luego definimos los métodos que manejarán esas excepciones.

```

1. @app.handle_errors(NotFound)
2. def notFoundHandler(self, request, failure):
3.     """
4.     Called when a 404 not found is raised.
5.     """
6.     return self.response(

```

```

7.         request, succeed=False, status=404, reason='Not
found.')
```

```

8.
9.     @app.handle_errors(MethodNotAllowed)
10.    def methodNotAllowedHandler(self, request, failure):
11.        """
12.        Called when a 405 is raised.
13.        """
14.        return self.response(
15.            request, succeed=False, status=405,
            reason="Method not allowed.")
```

Y confirmemos que esto sucede como esperamos:

```

1. >>> r = requests.put(url)
2. >>> r.json()
3. {'succeed': False, 'status': 405, 'reason': 'Method not
allowed.'}
4. >>> r = requests.get('http://localhost:7001/no-existe')
5. >>> r.json()
6. {'succeed': False, 'status': 404, 'reason': 'Not found.'}
```

## Base de datos

Veamos, ahora, cómo almacenar en una base de datos los correos que debe enviar nuestro servicio. Estaremos usando el motor de base de datos SQLite vía el módulo estándar `sqlite3` (<https://www.recuriospython.com/guias-y-manuales/base-de-datos-sqlite-con-sqlite3/>), junto con el módulo `adbapi` de Twisted. Este último nos permite acceder a una API sincrónica —la de SQLite— a través de una envoltura asincrónica; hemos hablado sobre ello en el artículo [Twisted: web y base de datos](https://www.recuriospython.com/guias-y-manuales/twisted-web-y-base-de-datos/) (<https://www.recuriospython.com/guias-y-manuales/twisted-web-y-base-de-datos/>).

Antes de eso, crearemos el *script* `initdb.py` que se encargará de inicializar la base de datos, esto es, crear el archivo y la tabla correspondiente.

```

1. #!/usr/bin/env python
2. # -*- coding: utf-8 -*-
3.
4. import sqlite3
5.
6. conn = sqlite3.connect('emailservice.db')
7. cursor = conn.cursor()
8. cursor.execute(
9.     'CREATE TABLE emails (id INTEGER PRIMARY KEY, recipient
TEXT, sent INTEGER);'
10. )
11. conn.commit()
12. conn.close()
```

Y ejecutémoslo una vez para que genere el archivo `emailservice.db`.



Ahora, volviendo a nuestra API web, el primer paso es importar la clase `adbapi.ConnectionPool`.

```
1. from twisted.enterprise.adbapi import ConnectionPool
```

Luego definiremos la conexión como un atributo.

```
1. class EmailService:
2.     app = Klein()
3.     conn = ConnectionPool('sqlite3', 'emailservice.db')
```

Queremos insertar una fila en nuestra tabla cada vez que se haga una petición `POST` a `/email`, para que funcione como un registro, y luego se pueda consultar su estado. Creemos el siguiente método para hacer lo primero.

```
1.     def insertEmail(self, transaction, to):
2.         """
3.         This function gets called by Twisted in a thread so
we can
4.         safely use blocking functions such as execute().
The return
5.         value is then retrieved to the main thread where
6.         runInteraction() was called.
7.         """
8.         transaction.execute(
9.             'INSERT INTO emails (recipient, sent) VALUES
(?, ?);',
10.            (to, False)
11.        )
12.        return transaction.lastrowid
```

El primer argumento `transaction` equivale a un *cursor* convencional (según el [estándar DB-API](https://www.recuriospython.com/guias-y-manuales/python-db-api-que-es-y-como-funciona/) (<https://www.recuriospython.com/guias-y-manuales/python-db-api-que-es-y-como-funciona/>)) para ejecutar consultas (provisto por Twisted), el segundo indica el destinatario del mensaje. Una vez ejecutada la consulta, retornamos el ID único asignado por SQLite.

Dentro de nuestro método `email()`, luego de la comprobación de los parámetros añadiremos lo siguiente.

```
1.         to = request.args[b'to'][0].decode('utf-8')
2.         # Run the insert query and return the inserted
ID.
3.         lastID = yield
self.conn.runInteraction(self.insertEmail, to)
4.         return self.response(request, emailID=lastID)
```

Esto hará que Twisted ejecute nuestra consulta en un hilo aparte para que no bloquee a nuestra API. La utilización de `yield` es similar a la del módulo estándar `asyncio`: Twisted atenderá otros



eventos mientras aguarda el resultado de la consulta. Nos resta, sin embargo, aplicar el decorador `inlineCallbacks`. Primero importémoslo:

```
1. from twisted.internet.defer import inlineCallbacks
```

Y después lo aplicamos, quedando nuestra función así:

```
1. @app.route('/email', methods=['GET', 'POST'])
2. @inlineCallbacks
3. def email(self, request):
4.     if request.method == b'POST':
5.         # Look for missing params.
6.         for param in (b'to', b'message', b'subject'):
7.             if param not in request.args:
8.                 return self.response(
9.                     request,
10.                     succeed=False,
11.                     status=400,
12.                     reason="Missing parameter:
13. '{}'.format(
14.                     param.decode('utf-8'))
15. to = request.args[b'to'][0].decode('utf-8')
16. # Run the insert query and return the inserted
17. ID.
18. lastID = yield
19. self.conn.runInteraction(self.insertEmail, to)
20. return self.response(request, emailID=lastID)
21.
22. elif request.method == b'GET':
23.     return 'Retornar lista de emails.'
```

Por último, chequeamos que esto funcione según lo planificado.

```
1. >>> params = {
2. ...     'to': 'nombre@ejemplo.com',
3. ...     'subject': 'Test',
4. ...     'message': '¡Hola, mundo!'
5. ... }
6. >>> r = requests.post(url, data=params)
7. >>> r.json()
8. {'succeed': True, 'status': 200, 'emailID': 1}
```

Esta es solo la mitad del trabajo. Resta la acción contraria: retornar la lista de todos los correos o bien alguno en particular, vía la petición `GET`. Para ello debemos redefinir la estructura de nuestro método para que acepte opcionalmente el ID de un mensaje.

```
1. @app.route('/email', methods=['GET', 'POST'],
2.             defaults={'emailID': None})
3. @app.route('/email/<int:emailID>', methods=['GET'])
4. @inlineCallbacks
5. def email(self, request, emailID):
```



Ahora definamos la lógica que obtendrá la información de la base de datos y la retornará al usuario como JSON.

```
1.         elif request.method == b'GET':
2.             if emailID is None:
3.                 # Get the full emails list.
4.                 emails = yield self.query(
5.                     'SELECT id, recipient, sent FROM
emails;')
6.             else:
7.                 # Get the specified email.
8.                 emails = yield self.query(
9.                     'SELECT id, recipient, sent FROM emails
WHERE id=?',
10.                    (emailID,)
11.                )
12.             if not emails:
13.                 raise NotFound
14.             emails = emails_to_json(emails)
15.             if emailID is not None:
16.                 emails = emails[0]
17.             response = self.response(request,
emails=emails)
18.             return response
```

Aquí hemos hecho referencia a dos objetos que aún no definimos, a saber, el método `query()`, que ejecuta una consulta:

```
1.     def query(self, *args):
2.         """Run a query using the current connection."""
3.         return self.conn.runQuery(*args)
```

Y la función `emails_to_json()` (por fuera de la clase), que convierte el resultado que proviene de SQLite a JSON:

```
1.     def emails_to_json(rows):
2.         def row_to_json(row):
3.             return json.dumps(
4.                 {'id': row[0], 'to': row[1], 'sent':
bool(row[2])})
5.         return tuple(map(row_to_json, rows))
```

Ahora las pruebas pertinentes:

```
1.     >>> r = requests.get(url)
2.     >>> r.json()
3.     {'succeed': True, 'status': 200, 'emails': [{'id': 1,
'to': 'nombre@ejemplo.com', 'sent': false}]}
4.     >>> r = requests.get(url + '/1')
5.     >>> r.json()
6.     {'succeed': True, 'status': 200, 'emails': '{"id": 1, "to":
"nombre@ejemplo.com", "sent": false}'}
```

## Enviando correos electrónicos



Comencemos por importar lo necesario: la función `sendmail()` de Twisted y la clase estándar `MIMEText` para construir el cuerpo del correo electrónico.

```
1. from email.mime.text import MIMEText
2. # [...]
3. from twisted.mail.smtp import sendmail
```

Luego modifiquemos el método `email()`, para que al recibir una petición POST envíe el mensaje una vez que lo almacenó en la base de datos.

```
1.         if request.method == b'POST':
2.             # [...]
3.             # Run the insert query and return the inserted
ID.
4.             lastID = yield
self.conn.runInteraction(self.insertEmail, to)
5.             message = MIMEText(
6.                 request.args[b'message'][0].decode("utf-
8"))
7.             message['Subject'] = \
8.                 request.args[b'subject'][0].decode("utf-8")
9.             message['From'] = USER
10.            message['To'] = to
11.            d = sendmail(
12.                EMAIL_SERVER, USER, [to], message,
13.                username=USER, password=PASSWORD
14.            )
15.            d.addCallback(self.emailSent, lastID)
16.            return self.response(request, emailID=lastID)
```

Deberemos crear las constantes `EMAIL_SERVER`, `USER` y `PASSWORD` con los datos del servidor SMTP correspondiente.

Y definamos el evento `emailSent()`, que será invocado cuando Twisted haya enviado el mensaje correctamente.

```
1.     def emailSent(self, result, emailID):
2.         """Called once the email was sent."""
3.         emails_sent = result[0]
4.         if emails_sent == 0:
5.             return
6.         self.query('UPDATE emails SET sent=1 WHERE id=?',
(emailID,))
```

Con estos arreglos nuestra API ya estará enviando correos electrónicos efectivamente.

```
1. # Enviar email.
2. >>> r = requests.post(url, data=params)
3. >>> r.json()
4. {'succeed': True, 'status': 200, 'emailID': 3}
5. # Consultar estado.
6. >>> r = requests.get(url + '/3')
7. >>> r.json()
```



```
8.    {'succeed': True, 'status': 200, 'emails': '{"id": 3, "to":  
      "nombre@ejemplo.com", "sent": true}'}
```

## Caché

Incorporaremos un pequeño sistema de caché en memoria, usando un diccionario.

```
1.    class EmailService:  
2.        app = Klein()  
3.        cache = {}  
4.        conn = ConnectionPool('sqlite3', 'emailservice.db')
```

Esto evitará que se ejecute la misma consulta múltiples veces al obtener la lista de mensajes o alguno en particular.

```
1.        elif request.method == b'GET':  
2.            # Look for a cached result before hitting the  
            database.  
3.            key = 'emails' if emailID is None else 'email-  
            {}'.format(emailID)  
4.            try:  
5.                return self.cache[key]  
6.            except KeyError:  
7.                pass  
8.            if emailID is None:  
9.                # Get the full emails list.  
10.               emails = yield self.query(  
11.                   'SELECT id, recipient, sent FROM  
            emails;')  
12.            else:  
13.                # Get the specified email.  
14.                emails = yield self.query(  
15.                   'SELECT id, recipient, sent FROM emails  
            WHERE id=?',  
16.                   (emailID,)  
17.                   )  
18.                if not emails:  
19.                    raise NotFound  
20.                emails = emails_to_json(emails)  
21.                if emailID is not None:  
22.                    emails = emails[0]  
23.                response = self.response(request,  
            emails=emails)  
24.                self.cache[key] = response  
25.                return response
```

## Código completo

```
1.    #!/usr/bin/env python  
2.    # -*- coding: utf-8 -*-  
3.  
4.    """  
5.        REST API example using Twisted Klein.  
6.        Copyright (C) 2018 Recursos Python -  
        reucrsospython.com.  
7.    """  
8.  
9.    from email.mime.text import MIMEText
```



```

10. import json
11.
12. from klein import Klein
13. from twisted.enterprise.adbapi import ConnectionPool
14. from twisted.internet.defer import inlineCallbacks
15. from twisted.mail.smtp import sendmail
16. from werkzeug.exceptions import NotFound, MethodNotAllowed
17.
18.
19. EMAIL_SERVER = ''
20. USER = ''
21. PASSWORD = ''
22.
23.
24. def emails_to_json(rows):
25.     def row_to_json(row):
26.         return json.dumps(
27.             {'id': row[0], 'to': row[1], 'sent':
bool(row[2])})
28.     return tuple(map(row_to_json, rows))
29.
30.
31. class EmailService:
32.     app = Klein()
33.     cache = {}
34.     conn = ConnectionPool('sqlite3', 'emailservice.db')
35.
36.     def query(self, *args):
37.         """Run a query using the current connection."""
38.         return self.conn.runQuery(*args)
39.
40.     def response(self, request, succeed=True, status=200,
**kwargs):
41.         """
42.         Build the response body as JSON and set the proper
content-type
43.         header.
44.         """
45.         request.setHeader('Content-Type',
'application/json')
46.         request.setResponseCode(status)
47.         return json.dumps(
48.             {'succeed': succeed, 'status': status,
**kwargs})
49.
50.     def emailSent(self, result, emailID):
51.         """Called once the email was sent."""
52.         emails_sent = result[0]
53.         if emails_sent == 0:
54.             return
55.         self.query('UPDATE emails SET sent=1 WHERE id=?',
(emailID,))
56.
57.     def insertEmail(self, transaction, to):
58.         """
59.         This function gets called by Twisted in a thread so
we can
60.         safely use blocking functions such as execute().
The return
61.         value is then retrieved to the main thread where
runInteraction() was called.
62.         """
63.         transaction.execute(
64.             'INSERT INTO emails (recipient, sent) VALUES
(?, ?);',
65.             (to, False)
66.         )
67.

```

```

68.         return transaction.lastrowid
69.
70.     @app.handle_errors(NotFound)
71.     def notFoundHandler(self, request, failure):
72.         """
73.         Called when a 404 not found is raised.
74.         """
75.         return self.response(
76.             request, succeed=False, status=404, reason='Not
found.')
77.
78.     @app.handle_errors(MethodNotAllowed)
79.     def methodNotAllowedHandler(self, request, failure):
80.         """
81.         Called when a 405 is raised.
82.         """
83.         return self.response(
84.             request, succeed=False, status=405,
reason="Method not allowed.")
85.
86.     @app.route('/email', methods=['GET', 'POST'],
87.                 defaults={'emailID': None})
88.     @app.route('/email/<int:emailID>', methods=['GET'])
89.     @inlineCallbacks
90.     def email(self, request, emailID):
91.         if request.method == b'POST':
92.             # Look for missing params.
93.             for param in (b'to', b'message', b'subject'):
94.                 if param not in request.args:
95.                     return self.response(
96.                         request,
97.                         succeed=False,
98.                         status=400,
99.                         reason="Missing parameter:
'{}'.format(
100.                             param.decode('utf-8'))
101.                         )
102.                 to = request.args[b'to'][0].decode('utf-8')
103.                 # Run the insert query and return the inserted
ID.
104.                 lastID = yield
self.conn.runInteraction(self.insertEmail, to)
105.                 message = MIMEText(
106.                     request.args[b'message'][0].decode("utf-
8"))
107.                 message['Subject'] = \
108.                     request.args[b'subject'][0].decode("utf-8")
109.                 message['From'] = USER
110.                 message['To'] = to
111.                 d = sendmail(
112.                     EMAIL_SERVER, USER, [to], message,
113.                     username=USER, password=PASSWORD
114.                 )
115.                 d.addCallback(self.emailSent, lastID)
116.                 return self.response(request, emailID=lastID)
117.
118.             elif request.method == b'GET':
119.                 # Look for a cached result before hitting the
database.
120.                 key = 'emails' if emailID is None else 'email-
{}'.format(emailID)
121.                 try:
122.                     return self.cache[key]
123.                 except KeyError:
124.                     pass
125.                 if emailID is None:
126.                     # Get the full emails list. ▲

```

```

127.         emails = yield self.query(
128.             'SELECT id, recipient, sent FROM
emails;')

129.     else:
130.         # Get the specified email.
131.         emails = yield self.query(
132.             'SELECT id, recipient, sent FROM emails
WHERE id=?',
133.             (emailID,)
134.             )
135.         if not emails:
136.             raise NotFound
137.         emails = emails_to_json(emails)
138.         if emailID is not None:
139.             emails = emails[0]
140.         response = self.response(request,
emails=emails)
141.         self.cache[key] = response
142.         return response
143.
144.
145. if __name__ == '__main__':
146.     emailservice = EmailService()
147.     emailservice.app.run('localhost', 7001)

```

## Artículos relacionados

- [Tareas en segundo plano con PyQt/PySide](https://recursospython.com/guias-y-manuales/tareas-en-segundo-plano-con-pyqt/)  
(<https://recursospython.com/guias-y-manuales/tareas-en-segundo-plano-con-pyqt/>)
- [Chat vía web con WebSockets y Twisted](https://recursospython.com/codigos-de-fuente/chat-via-web-websockets-twisted/)  
(<https://recursospython.com/codigos-de-fuente/chat-via-web-websockets-twisted/>)
- [Twisted: web y base de datos](https://recursospython.com/guias-y-manuales/twisted-web-y-base-de-datos/)  
(<https://recursospython.com/guias-y-manuales/twisted-web-y-base-de-datos/>)
- [Twisted - Arquitectura del framework de red más popular](https://recursospython.com/guias-y-manuales/twisted-arquitectura-del-framework-de-red-mas-popular/)  
(<https://recursospython.com/guias-y-manuales/twisted-arquitectura-del-framework-de-red-mas-popular/>)
- [Introducción a Twisted y al desarrollo de servidores](https://recursospython.com/guias-y-manuales/introduccion-a-twisted/)  
(<https://recursospython.com/guias-y-manuales/introduccion-a-twisted/>)

Donar 

¿Te gusta nuestro contenido? ¡Ayudanos a [seguir creciendo con una donación \(/donar/\)](#)!

Entrada publicada en  
Códigos de fuente (<https://recursospython.com/category/codigos-de-fuente/>)

Guías y Manuales (<https://recursospython.com/category/guias-y-manuales/>) con las



etiquetas [klein \(https://recursospython.com/tag/klein/\)](https://recursospython.com/tag/klein/)

[twisted \(https://recursospython.com/tag/twisted/\)](https://recursospython.com/tag/twisted/)

◄ [Introducción a web2py \(https://recursospython.com/guias-y-manuales/introduccion-a-web2py/\)](https://recursospython.com/guias-y-manuales/introduccion-a-web2py/)

[Caja de texto con menú de copiar, cortar y pegar en Tcl/Tk \(tkinter\)](#)

► [\(https://recursospython.com/codigos-de-fuente/caja-de-texto-con-menu-de-copiar-cortar-pegar/\)](https://recursospython.com/codigos-de-fuente/caja-de-texto-con-menu-de-copiar-cortar-pegar/)

---

## Deja una respuesta

Comentario \*

Nombre \*

Email \*

**Publicar el comentario**

© 2013 - 2023

[¡Suscríbete a nuestra newsletter! \(https://www.recursospython.com/newsletter/\)](https://www.recursospython.com/newsletter/)

[Políticas de Uso y Privacidad \(https://www.recursospython.com/politicas-de-uso-y-privacidad/\)](https://www.recursospython.com/politicas-de-uso-y-privacidad/)

En inglés: [Python Assets \(https://pythonassets.com/\)](https://pythonassets.com/)



[\(https://creativecommons.org/licenses/by-nc/3.0/deed.es\)](https://creativecommons.org/licenses/by-nc/3.0/deed.es)

