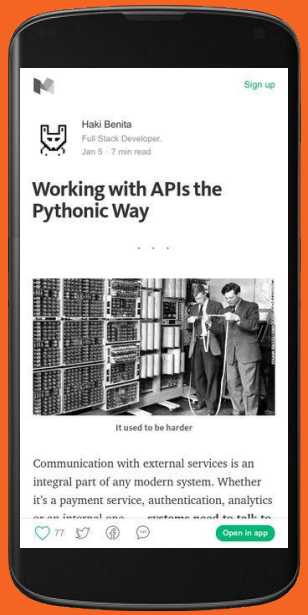
Working with APIs The Pythonic Way



Haki Benita
hakibenita@gmail.com

Systems need to talk

So let's make them talk!

A Payment API

```
POST https://foo-payments.com/api/charge
{
    “token”: <string>,
    “amount”: <number>,
    “uid”: <string>
}
```

A Payment API - Success

```
200 OK
{
    “uid”: <string>,
    “amount”: <number>,
    “token”: <string>,
    “expiration”: <string, isoformat>,
    “transaction_id”: <number>
}
```

A Payment API - Failure

```
400 Bad Request
{
    “uid”: <string>,
    “error”: <number>,
    “message”: <string>
}
```

Error codes

1 = Refused

2 = Stolen

The simple implmentation

payments.py

```
def charge(amount, token):
    headers = {
        "Authorization": "Bearer " + TOKEN,
    }

    payload = {
        "token": token,
        "amount": amount,
        "uid": str(uuid.uuid4()),
    }

    response = requests.post(
        BASE_URL + '/charge',
        json=payload,
        headers=headers,
    )
    response.raise_for_status()

    return response.json()
```

Most developers will stop here.

What is the problem?

Errors

- **Connection**
  Remote service is unavailable.

- **Timeout**
  The remote is taking too long and we might choke.

- **Application Errors**
  How are they represented?

Handling Errors

To provide a complete API

Our module must communicate errors!

Types of Errors

- **Connection errors**
  Timeout or connection refused.

- **Application errors**
  Refusal or stolen card.

- **Log what you can't handle**
  500 for example.

errors.py

```
class Error(Exception):
    pass

class Unavailable(Error):
    pass

class PaymentGatewayError(Error):
    def __init__(self, code, message):
        self.code = code
        self.message = message

class Refused(PaymentGatewayError):
    pass

class Stolen(PaymentGatewayError):
    pass
```

A base Error class

Catch all errors from a certain module:

```
from payment.errors import Error as PaymentError

try:
  # Do something...
except PaymentError:
  # Handle errors from payment
```

payments.py

- Handle errors

```
import logging
from . import errors
from requests import ConnectionError, Timeout

logger = logging.getLogger('payments')

def charge(amount, token, timeout=5):

    ...

    try:
        response = requests.post(...)
        response.raise_for_status()

    except (ConnectionError, Timeout) as e:
        raise errors.Unavailable() from e

    except requests.exceptions.HTTPError as e:
        ( next page )
```

payments.py

- Handle errors

```
except requests.exceptions.HTTPError as e:

    if e.response.status_code == 400:
        error = e.response.json()
        code = error['code']
        message = error['message']

        if code == 1:
            raise Refused(code, message) from e

        elif code == 2:
            raise Stolen(code, message) from e

        else:
            raise PaymentGatewayError(code, message) from e

    logger.exception("Payment service had internal error")
    raise Unavailable() from e
```

Handling Errors

- User does not need to handle requests exceptions.
  Requests is an implementation detail.

- Errors are communicated as exceptions.
  No need to parse direct response from API.

- Errors can be handled in different ways
  We can retry on connection error for example...

The Response

- Our function returns a dict.

- But, we know what we are going to get!

Namedtuple

A tuple, with names:

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 25)
p.x  # 10
```

payments.py

- Handle errors
- Define the response

```
from collections import namedtuple

ChargeResponse = namedtuple('ChargeResponse', [
    'uid',
    'amount',
    'token',
    'expiration',
    'transaction_id',
])
```

payments.py

- Handle errors
- Define the response

```
...

data = response.json()

expiration = datetime.strptime(
    data['expiration'],
    "%Y-%m-%dT%H:%M:%S.%f",
)

return ChargeResponse(
    uid=uuid.UID(data['uid']),
    amount=data['amount'],
    token=data['token'],
    expiration=expiration,
    transaction_id=data['transaction_id'],
)
```

The Response

- Our function now returns a `ChargeResponse` object.

- Validation and casting can be added easily.

- Eliminate dependency on the API serialization format.

Quiz

Rank the following data structures by memory efficiency:

- Dict
- List
- Tuple
- Namedtuple
- Class
- Class with slots

Answer

1. Tuple
2. Namedtuple
3. Class with slots
4. List
5. Dict
6. Class

A Session

- We keep posting to the same host.
- “Requests” session uses a connection pool internally.
- Add useful configuration such as blocking cookies.

payments.py

- Handle errors
- Define the response
- Use a session

```
import http.cookiejar


class BlockAll(http.cookiejar.CookiePolicy):
    def set_ok(self, cookie, request):
        return False


payment_session = requests.Session()
payment_session.cookies.policy = BlockAll()
```

payments.py

- Handle errors
- Define the response
- Use a session

```
...

def charge(
    amount,
    token,
    timeout=5,
):

    ...

    response = payment_session.post(...)

    ...
```

More Actions!

- Refund for example.
- Authentication is the same.
- Transport is the same.
- Connection errors handled the same.

payments.py

- Handle errors
- Define the response
- Use a session
- Refactor

```
def make_payment_request(path, payload, timeout=5):
    headers = {
        "Authorization": "Bearer " + TOKEN,
    }

    try:
        response = payment_session.post(
            BASE_URL + path,
            json=payload,
            headers=headers,
            timeout=timeout,
        )
    except (ConnectionError, Timeout) as e:
        raise errors.Unavailable() from e

    response.raise_for_status()

    return response.json()
```

payments.py

- Handle errors
- Define the response
- Use a session
- Refactor

```
def charge(amount, token):
    try:
        data = make_payment_request('/charge', {
            'uid': str(uuid.uuid4()),
            'amount': amount,
            'token': token,
        })

    except requests.HTTPError as e:
        # Handle charge errors...

    return ChargeResponse(...)
```

payments.py

- Handle errors
- Define the response
- Use a session
- Refactor
- More actions

```
RefundResponse = namedtuple('RefundResponse', [
    'transaction_id',
    'refund_id',
])

def refund(transaction_id):
    try:
        data = make_payment_request('/refund', {
            'uid': str(uuid.uuid4()),
            'transaction_id': transaction_id,
        })
    except requests.HTTPError as e:
        # TODO: Handle refund remote errors

    return RefundResponse(
        'transaction_id': data['transaction_id'],
        'refund_id': data['refund_transation_id'],
    )
```

Looks good

So are we done???

NO!

Testing

- Focus on testing code that's using our module.

- Can't make calls to *real* external service during tests.

Our module has a simple interface so it's <u>easy to mock!</u>

payments.py

- Handle errors
- Define the response
- Use a session
- Refactor
- More actions
- Test

```
from unittest import TestCase
from unittest.mock import patch
from payment.payment import ChargeResponse

def TestApp(TestCase):

    @mock.patch('payment.charge')
    def test_success(self, mock_charge):
        self.assertEqual(user.charged_transactions, 0)
        mock_charge.return_value = ChargeResponse(
            uid=’test-uid’,
            amount=100,
            token=’test-token’,
            expiration=datetime.today(),
            transaction_id=12345,
        )
        charge_user(user, 100)
        self.assertEqual(user.charged_transactions, 1)
```

payments.py

- Handle errors
- Define the response
- Use a session
- Refactor
- More actions
- Test

```
from payment import errors

def TestApp(TestCase):

    @mock.patch('payment.charge')
    def test_failure(self, mock_charge):
        mock_charge.side_effect = errors.Stolen
        self.assertEqual(user.charged_transactions, 0)

        charge_user(user, 100)

        self.assertEqual(user.charged_transactions, 0)
```

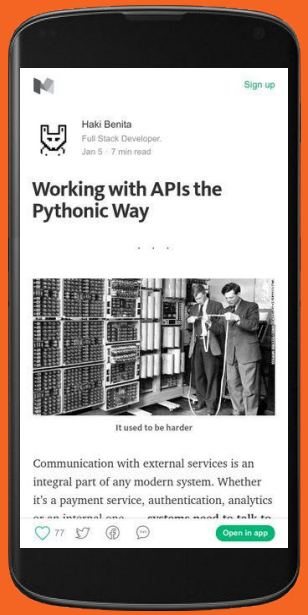
Challenges

- **Failures** (connection, application…)
- **How to handle errors** (propagate, retry, report…)
- **Transport** (http, ws, files…)
- **Serialization format** (json, xml, msgpack…)
- **Code reuse**
- **Testing**

What we did

- Naive implementation
- Handled errors
- Defined the response
- Used a session
- Refactored
- Added more actions
- Test!

Thank you for listening!



Haki Benita
hakibenita@gmail.com
medium.com/@hakibenita
twitter.com/be_haki