Crochet Documentation

Release 2.1.1+0.g9ca30d0

Itamar Turner-Trauring

Jul 01, 2023

Contents

1 Table of Contents **3**

1.1 Introduction . . . 3
1.1.1 Crochet: Use Twisted anywhere! . . . 3
1.1.2 API and features . . . 3
1.1.3 Examples . . . 4
1.2 Using Crochet . . . 9
1.2.1 Setup . . . 9
1.2.2 @wait_for: Blocking calls into Twisted . . . 9
1.2.3 @run_in_reactor: Asynchronous results . . . 11
1.2.4 Using Crochet from Twisted applications . . . 13
1.2.5 Unit testing . . . 13
1.3 Best Practices . . . 14
1.3.1 Hide Twisted and Crochet . . . 14
1.3.2 Minimize decorated code . . . 16
1.4 Static Type Checking . . . 16
1.5 Known Issues and Workarounds . . . 17
1.5.1 Don't Call Twisted APIs from non-Twisted threads . . . 17
1.5.2 Preventing deadlocks on shutdown . . . 17
1.5.3 Reducing Twisted log messages . . . 18
1.5.4 Missing tracebacks . . . 18
1.5.5 uWSGI, multiprocessing, Celery . . . 18
1.6 Differences from async/await . . . 19
1.7 API Reference . . . 19
1.8 What's New . . . 20
1.8.1 2.1.0 . . . 20
1.8.2 2.0.0 . . . 20
1.8.3 1.12.0 . . . 20
1.8.4 1.11.0 . . . 21
1.8.5 1.10.0 . . . 21
1.8.6 1.9.0 . . . 21
1.8.7 1.8.0 . . . 21
1.8.8 1.7.0 . . . 21
1.8.9 1.6.0 . . . 22
1.8.10 1.5.0 . . . 22
1.8.11 1.4.0 . . . 22
1.8.12 1.3.0 . . . 22

1.8.13 1.2.0 . . . 22
1.8.14 1.1.0 . . . 23
1.8.15 1.0.0 . . . 23
1.8.16 0.9.0 . . . 23
1.8.17 0.8.1 . . . 24
1.8.18 0.7.0 . . . 24
1.8.19 0.6.0 . . . 24
1.8.20 0.5.0 . . . 24

Index **25**

Crochet is an MIT-licensed library that makes it easier for blocking and threaded applications like Flask or Django to use the Twisted networking framework.

Here's an example of a program using Crochet. Notice that you get a completely blocking interface to Twisted and do not need to run the Twisted reactor, the event loop, yourself.

```
#!/usr/bin/python
"""
Do a DNS lookup using Twisted's APIs.
"""
from __future__ import print_function

# The Twisted code we'll be using:
from twisted.names import client

from crochet import setup, wait_for
setup()


# Crochet layer, wrapping Twisted's DNS library in a blocking call.
@wait_for(timeout=5.0)
def gethostbyname(name):
    """Lookup the IP of a given hostname.

    Unlike socket.gethostbyname() which can take an arbitrary amount of time
    to finish, this function will raise crochet.TimeoutError if more than 5
    seconds elapse without an answer being received.
    """
    d = client.lookupAddress(name)
    d.addCallback(lambda result: result[0][0].payload.dottedQuad())
    return d


if __name__ == '__main__':
    # Application code using the public API - notice it works in a normal
    # blocking manner, with no event loop visible:
    import sys
    name = sys.argv[1]
    ip = gethostbyname(name)
    print(name, "->", ip)
```

Run on the command line:

```
$ python blockingdns.py twistedmatrix.com
twistedmatrix.com -> 66.35.39.66
```

You can also wrap `async` functions. Here is the equivalent code to the previous example, but using an `async/await` function:

```
@wait_for(timeout=5.0)
async def gethostbyname(name):
    result = await client.lookupAddress(name)
    return result[0][0].payload.dottedQuad()
```

CHAPTER 1

Table of Contents

1.1 Introduction

1.1.1 Crochet: Use Twisted anywhere!

Crochet is an MIT-licensed library that makes it easier to use Twisted from regular blocking code. Some use cases include:

- Easily use Twisted from a blocking framework like Django or Flask.
- Write a library that provides a blocking API, but uses Twisted for its implementation.
- Port blocking code to Twisted more easily, by keeping a backwards compatibility layer.
- Allow normal Twisted programs that use threads to interact with Twisted more cleanly from their threaded parts. For example, this can be useful when using Twisted as a WSGI container.

Crochet is maintained by Itamar Turner-Trauring.

> **Note:** Crochet development is pretty slow these days because mostly it **Just Works**. PyPI shows about 30,000 downloads a month, so existing users seem happy: https://pypistats.org/packages/crochet

You can install Crochet by running:

```
$ pip install crochet
```

Downloads are available on PyPI.

Documentation can be found on Read The Docs.

Bugs and feature requests should be filed at the project Github page.

1.1.2 API and features

Crochet supports Python 3.8, 3.9, 3.10, and 3.11 as well as PyPy3.

Crochet provides the following basic APIs:

- Allow blocking code to call into Twisted and block until results are available or a timeout is hit, using the `crochet.wait_for` decorator.
- A lower-level API (`crochet.run_in_reactor`) allows blocking code to run code "in the background" in the Twisted thread, with the ability to repeatedly check if it's done.

Crochet will do the following on your behalf in order to enable these APIs:

- Transparently start Twisted's reactor in a thread it manages.
- Shut down the reactor automatically when the process' main thread finishes.
- Hook up Twisted's log system to the Python standard library `logging` framework. Unlike Twisted's built-in `logging` bridge, this includes support for blocking *Handler* instances.

1.1.3 Examples

Background scheduling

You can use Crochet to schedule events that will run in the background without slowing down the page rendering of your web applications:

```
#!/usr/bin/python
"""
An example of scheduling time-based events in the background.

Download the latest EUR/USD exchange rate from Yahoo every 30 seconds in the
background; the rendered Flask web page can use the latest value without
having to do the request itself.

Note this is example is for demonstration purposes only, and is not actually
used in the real world. You should not do this in a real application without
reading Yahoo's terms-of-service and following them.
"""

from __future__ import print_function

from flask import Flask

from twisted.internet.task import LoopingCall
from twisted.web.client import getPage
from twisted.python import log

from crochet import wait_for, run_in_reactor, setup
setup()


# Twisted code:
class _ExchangeRate(object):
    """Download an exchange rate from Yahoo Finance using Twisted."""

    def __init__(self, name):
        self._value = None
        self._name = name

    # External API:
```

(continues on next page)

(continued from previous page)

```
    def latest_value(self):
        """Return the latest exchange rate value.

        May be None if no value is available.
        """
        return self._value

    def start(self):
        """Start the background process."""
        self._lc = LoopingCall(self._download)
        # Run immediately, and then every 30 seconds:
        self._lc.start(30, now=True)

    def _download(self):
        """Download the page."""
        print("Downloading!")
        def parse(result):
            print("Got %r back from Yahoo." % (result,))
            values = result.strip().split(",")
            self._value = float(values[1])
        d = getPage(
            "http://download.finance.yahoo.com/d/quotes.csv?e=.csv&f=c4l1&s=%s=X"
            % (self._name,))
        d.addCallback(parse)
        d.addErrback(log.err)
        return d


# Blocking wrapper:
class ExchangeRate(object):
    """Blocking API for downloading exchange rate."""

    def __init__(self, name):
        self._exchange = _ExchangeRate(name)

    @run_in_reactor
    def start(self):
        self._exchange.start()

    @wait_for(timeout=1)
    def latest_value(self):
        """Return the latest exchange rate value.

        May be None if no value is available.
        """
        return self._exchange.latest_value()


EURUSD = ExchangeRate("EURUSD")
app = Flask(__name__)

@app.route('/')
def index():
    rate = EURUSD.latest_value()
    if rate is None:
        rate = "unavailable, please refresh the page"
    return "Current EUR/USD exchange rate is %s." % (rate,)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    import sys, logging
    logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
    EURUSD.start()
    app.run()
```

SSH into your server

You can SSH into your Python process and get a Python prompt, allowing you to poke around in the internals of your running program:

```
#!/usr/bin/python
"""
A demonstration of Conch, allowing you to SSH into a running Python server and
inspect objects at a Python prompt.

If you're using the system install of Twisted, you may need to install Conch
separately, e.g. on Ubuntu:

   $ sudo apt-get install python-twisted-conch

Once you've started the program, you can ssh in by doing:

    $ ssh admin@localhost -p 5022

The password is 'secret'. Once you've reached the Python prompt, you have
access to the app object, and can import code, etc.:

    >>> 3 + 4
    7
    >>> print(app)
    <flask.app.Flask object at 0x18e1690>

"""

import logging

from flask import Flask
from crochet import setup, run_in_reactor
setup()

# Web server:
app = Flask(__name__)

@app.route('/')
def index():
    return "Welcome to my boring web server!"


@run_in_reactor
def start_ssh_server(port, username, password, namespace):
    """
    Start an SSH server on the given port, exposing a Python prompt with the
```

(continues on next page)

(continued from previous page)

```
    given namespace.
    """
    # This is a lot of boilerplate, see http://tm.tl/6429 for a ticket to
    # provide a utility function that simplifies this.
    from twisted.internet import reactor
    from twisted.conch.insults import insults
    from twisted.conch import manhole, manhole_ssh
    from twisted.cred.checkers import (
        InMemoryUsernamePasswordDatabaseDontUse as MemoryDB)
    from twisted.cred.portal import Portal

    sshRealm = manhole_ssh.TerminalRealm()
    def chainedProtocolFactory():
        return insults.ServerProtocol(manhole.Manhole, namespace)
    sshRealm.chainedProtocolFactory = chainedProtocolFactory

    sshPortal = Portal(sshRealm, [MemoryDB(**{username: password})])
    reactor.listenTCP(port, manhole_ssh.ConchFactory(sshPortal),
                      interface="127.0.0.1")


if __name__ == '__main__':
    import sys
    logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
    start_ssh_server(5022, "admin", "secret", {"app": app})
    app.run()
```

DNS query

Twisted also has a fully featured DNS library:

```
#!/usr/bin/python
"""
A command-line application that uses Twisted to do an MX DNS query.
"""

from __future__ import print_function

from twisted.names.client import lookupMailExchange
from crochet import setup, wait_for
setup()


# Twisted code:
def _mx(domain):
    """
    Return Deferred that fires with a list of (priority, MX domain) tuples for
    a given domain.
    """
    def got_records(result):
        return sorted(
            [(int(record.payload.preference), str(record.payload.name))
             for record in result[0]])
    d = lookupMailExchange(domain)
    d.addCallback(got_records)
```

(continues on next page)

(continued from previous page)

```
    return d


# Blocking wrapper:
@wait_for(timeout=5)
def mx(domain):
    """
    Return list of (priority, MX domain) tuples for a given domain.
    """
    return _mx(domain)


# Application code:
def main(domain):
    print("Mail servers for %s:" % (domain,))
    for priority, mailserver in mx(domain):
        print(priority, mailserver)


if __name__ == '__main__':
    import sys
    main(sys.argv[1])
```

Using Crochet in normal Twisted code

You can use Crochet's APIs for calling into the reactor thread from normal Twisted applications:

```
#!/usr/bin/python
"""
An example of using Crochet from a normal Twisted application.
"""

import sys

from crochet import no_setup, wait_for
# Tell Crochet not to run the reactor:
no_setup()

from twisted.internet import reactor
from twisted.python import log
from twisted.web.wsgi import WSGIResource
from twisted.web.server import Site
from twisted.names import client

# A WSGI application, will be run in thread pool:
def application(environ, start_response):
    start_response('200 OK', [])
    try:
        ip = gethostbyname('twistedmatrix.com')
        return "%s has IP %s" % ('twistedmatrix.com', ip)
    except Exception, e:
        return 'Error doing lookup: %s' % (e,)

# A blocking API that will be called from the WSGI application, but actually
# uses DNS:
```

(continues on next page)

(continued from previous page)

```
@wait_for(timeout=10)
def gethostbyname(name):
    d = client.lookupAddress(name)
    d.addCallback(lambda result: result[0][0].payload.dottedQuad())
    return d

# Normal Twisted code, serving the WSGI application and running the reactor:
def main():
    log.startLogging(sys.stdout)
    pool = reactor.getThreadPool()
    reactor.listenTCP(5000, Site(WSGIResource(reactor, pool, application)))
    reactor.run()

if __name__ == '__main__':
    main()
```

1.2 Using Crochet

Using Crochet involves three parts: reactor setup, defining functions that call into Twisted's reactor, and using those functions.

1.2.1 Setup

Crochet does a number of things for you as part of setup. Most significantly, it runs Twisted's reactor in a thread it manages. Doing setup is easy, just call the `setup()` function:

```
from crochet import setup
setup()
```

Since Crochet is intended to be used as a library, multiple calls work just fine; if more than one library does `crochet.setup()` only the first one will do anything.

1.2.2 @wait_for: Blocking calls into Twisted

Now that you've got the reactor running, the next stage is defining some functions that will run inside the Twisted reactor thread. Twisted's APIs are not thread-safe, and so they cannot be called directly from another thread. Moreover, results may not be available immediately. The easiest way to deal with these issues is to decorate a function that calls Twisted APIs with `crochet.wait_for`.

- When the decorated function is called, the code will not run in the calling thread, but rather in the reactor thread.
- The function blocks until a result is available from the code running in the Twisted thread. The returned result is the result of running the code; if the code throws an exception, an exception is thrown.
- If the underlying code returns a `Deferred`, it is handled transparently; its results are extracted and passed to the caller.
- `crochet.wait_for` takes a `timeout` argument, a `float` indicating the number of seconds to wait until a result is available. If the given number of seconds pass and the underlying operation is still unfinished a `crochet.TimeoutError` exception is raised, and the wrapped `Deferred` is canceled. If the underlying API supports cancellation this might free up any unused resources, close outgoing connections etc., but cancellation is not guaranteed and should not be relied on.

To see what this means, let's return to the first example in the documentation:

```
#!/usr/bin/python
"""
Do a DNS lookup using Twisted's APIs.
"""
from __future__ import print_function

# The Twisted code we'll be using:
from twisted.names import client

from crochet import setup, wait_for
setup()


# Crochet layer, wrapping Twisted's DNS library in a blocking call.
@wait_for(timeout=5.0)
def gethostbyname(name):
    """Lookup the IP of a given hostname.

    Unlike socket.gethostbyname() which can take an arbitrary amount of time
    to finish, this function will raise crochet.TimeoutError if more than 5
    seconds elapse without an answer being received.
    """
    d = client.lookupAddress(name)
    d.addCallback(lambda result: result[0][0].payload.dottedQuad())
    return d


if __name__ == '__main__':
    # Application code using the public API - notice it works in a normal
    # blocking manner, with no event loop visible:
    import sys
    name = sys.argv[1]
    ip = gethostbyname(name)
    print(name, "->", ip)
```

Twisted's `lookupAddress()` call returns a `Deferred`, but the code calling the decorated `gethostbyname()` doesn't know that. As far as the caller concerned is just calling a blocking function that returns a result or raises an exception. Run on the command line with a valid domain we get:

```
$ python blockingdns.py twistedmatrix.com
twistedmatrix.com -> 66.35.39.66
```

If we try to call the function with an invalid domain, we get back an exception:

```
$ python blockingdns.py doesnotexist
Trace back (most recent call last):
  File "examples/blockingdns.py", line 33, in <module>
    ip = gethostbyname(name)
  File "/home/itamar/crochet/crochet/_eventloop.py", line 434, in wrapper
    return eventual_result.wait(timeout)
  File "/home/itamar/crochet/crochet/_eventloop.py", line 216, in wait
    result.raiseException()
  File "<string>", line 2, in raiseException
twisted.names.error.DNSNameError: <Message id=36791 rCode=3 maxSize=0 flags=answer,
↪recDes,recAv queries=[Query('doesnotexist', 1, 1)] authority=[<RR name= type=SOA␣
↪class=IN ttl=1694s auth=False>]>
```

(continues on next page)

(continued from previous page)

You can, similarly, wrap an `async` function:

```
#!/usr/bin/python
"""
Async/await DNS lookup using Twisted's APIs.
"""
from __future__ import print_function

# The Twisted code we'll be using:
from twisted.names import client

from crochet import setup, wait_for
setup()


# Crochet layer, wrapping Twisted's DNS library in a blocking call.
# Uses async/await.
@wait_for(timeout=5.0)
async def gethostbyname(name):
    """Lookup the IP of a given hostname.

    Unlike socket.gethostbyname() which can take an arbitrary amount of time
    to finish, this function will raise crochet.TimeoutError if more than 5
    seconds elapse without an answer being received.
    """
    result = await client.lookupAddress(name)
    return result[0][0].payload.dottedQuad()


if __name__ == '__main__':
    # Application code using the public API - notice it works in a normal
    # blocking manner, with no event loop visible:
    import sys
    name = sys.argv[1]
    ip = gethostbyname(name)
    print(name, "->", ip)
```

1.2.3 @run_in_reactor: Asynchronous results

`wait_for` is implemented using `run_in_reactor`, a more sophisticated and lower-level API. Rather than waiting until a result is available, it returns a special object supporting multiple attempts to retrieve results, as well as manual cancellation. This can be useful for running tasks "in the background", i.e. asynchronously, as opposed to blocking and waiting for them to finish.

Decorating a function that calls Twisted APIs with `run_in_reactor` has two consequences:

- When the function is called, the code will not run in the calling thread, but rather in the reactor thread.
- The return result from a decorated function is an `EventualResult` instance, wrapping the result of the underlying code, with built-in support for functions that return `Deferred` instances as well as `async` functions.

`EventualResult` has the following basic methods:

- `wait(timeout)`: Return the result when it becomes available; if the result is an exception it will be

raised. The timeout argument is a `float` indicating a number of seconds; `wait()` will throw `crochet.TimeoutError` if the timeout is hit.

- `cancel()`: Cancel the operation tied to the underlying `Deferred`. Many, but not all, `Deferred` results returned from Twisted allow the underlying operation to be canceled. Even if implemented, cancellation may not be possible for a variety of reasons, e.g. it may be too late. Its main purpose to free up no longer used resources, and it should not be relied on otherwise.

There are also some more specialized methods:

- `original_failure()` returns the underlying Twisted Failure object if your result was a raised exception, allowing you to print the original traceback that caused the exception. This is necessary because the default exception you will see raised from `EventualResult.wait()` won't include the stack from the underlying Twisted code where the exception originated.
- `stash()`: Sometimes you want to store the `EventualResult` in memory for later retrieval. This is specifically useful when you want to store a reference to the `EventualResult` in a web session like Flask's (see the example below). `stash()` stores the `EventualResult` in memory, and returns an integer uid that can be used to retrieve the result using `crochet.retrieve_result(uid)`. Note that retrieval works only once per uid. You will need the stash the `EventualResult` again (with a new resulting uid) if you want to retrieve it again later.

In the following example, you can see all of these APIs in use. For each user session, a download is started in the background. Subsequent page refreshes will eventually show the downloaded page.

```
#!/usr/bin/python
"""
A flask web application that downloads a page in the background.
"""

import logging
from flask import Flask, session, escape
from crochet import setup, run_in_reactor, retrieve_result, TimeoutError

# Can be called multiple times with no ill-effect:
setup()

app = Flask(__name__)


@run_in_reactor
def download_page(url):
    """
    Download a page.
    """
    from twisted.web.client import getPage
    return getPage(url)


@app.route('/')
def index():
    if 'download' not in session:
        # Calling an @run_in_reactor function returns an EventualResult:
        result = download_page('http://www.google.com')
        session['download'] = result.stash()
        return "Starting download, refresh to track progress."

    # Retrieval is a one-time operation, so the uid in the session cannot be
    # reused:
```

(continues on next page)

(continued from previous page)

```
    result = retrieve_result(session.pop('download'))
    try:
        download = result.wait(timeout=0.1)
        return "Downloaded: " + escape(download)
    except TimeoutError:
        session['download'] = result.stash()
        return "Download in progress..."
    except:
        # The original traceback of the exception:
        return "Download failed:\n" + result.original_failure().getTraceback()


if __name__ == '__main__':
    import os, sys
    logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
    app.secret_key = os.urandom(24)
    app.run()
```

1.2.4 Using Crochet from Twisted applications

If your application is already planning on running the Twisted reactor itself (e.g. you're using Twisted as a WSGI container), Crochet's default behavior of running the reactor in a thread is a problem. To solve this, Crochet provides the `no_setup()` function, which causes future calls to `setup()` to do nothing. Thus, an application that will run the Twisted reactor but also wants to use a Crochet-using library must run it first:

```
from crochet import no_setup
no_setup()
# Only now do we import libraries that might run crochet.setup():
import blockinglib

# ... setup application ...

from twisted.internet import reactor
reactor.run()
```

1.2.5 Unit testing

Both `@wait_for` and `@run_in_reactor` expose the underlying Twisted function via a `__wrapped__` attribute. This allows unit testing of the Twisted code without having to go through the Crochet layer.

```
#!/usr/bin/python
"""
Demonstration of accessing wrapped functions for testing.
"""

from __future__ import print_function

from crochet import setup, run_in_reactor
setup()

@run_in_reactor
def add(x, y):
    return x + y
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    print("add(1, 2) returns EventualResult:")
    print("    ", add(1, 2))
    print("add.__wrapped__(1, 2) is the result of the underlying function:")
    print("    ", add.__wrapped__(1, 2))
```

When run, this gives the following output:

```
add(1, 2) returns EventualResult:
     <crochet._eventloop.EventualResult object at 0x2e8b390>
add.__wrapped__(1, 2) returns result of underlying function:
     3
```

1.3 Best Practices

1.3.1 Hide Twisted and Crochet

Consider some synchronous do-one-thing-after-the-other application code that wants to use event-driven Twisted-using code. We have two threads at a minimum: the application thread(s) and the reactor thread. There are also multiple layers of code involved in this interaction:

- **Twisted code:** Should only be called in reactor thread. This may be code from the Twisted package itself, or more likely code you have written that is built on top of Twisted.
- **@wait_for/@run_in_reactor wrappers:** The body of the functions runs in the reactor thread... but the caller should be in the application thread.
- **The application code:** Runs in the application thread(s), expects synchronous/blocking calls.

Sometimes the first two layers suffice, but there are some issues with only having these. First, if you're using `@run_in_reactor` it requires the application code to understand Crochet's API, i.e. `EventualResult` objects. Second, if the wrapped function returns an object that expects to interact with Twisted, the application code will not be able to use that object since it will be called in the wrong thread.

A better solution is to have an additional layer in-between the application code and `@wait_for`/`@run_in_reactor` wrappers. This layer can hide the details of the Crochet API and wrap returned Twisted objects if necessary. As a result the application code simply seems a normal API, with no need to understand `EventualResult` objects or Twisted.

In the following example the different layers of the code demonstrate this separation: `_ExchangeRate` is just Twisted code, and `ExchangeRate` provides a blocking wrapper using Crochet.

```
#!/usr/bin/python
"""
An example of scheduling time-based events in the background.

Download the latest EUR/USD exchange rate from Yahoo every 30 seconds in the
background; the rendered Flask web page can use the latest value without
having to do the request itself.

Note this is example is for demonstration purposes only, and is not actually
used in the real world. You should not do this in a real application without
reading Yahoo's terms-of-service and following them.
```

(continues on next page)

(continued from previous page)

```
"""

from __future__ import print_function

from flask import Flask

from twisted.internet.task import LoopingCall
from twisted.web.client import getPage
from twisted.python import log

from crochet import wait_for, run_in_reactor, setup
setup()


# Twisted code:
class _ExchangeRate(object):
    """Download an exchange rate from Yahoo Finance using Twisted."""

    def __init__(self, name):
        self._value = None
        self._name = name

    # External API:
    def latest_value(self):
        """Return the latest exchange rate value.

        May be None if no value is available.
        """
        return self._value

    def start(self):
        """Start the background process."""
        self._lc = LoopingCall(self._download)
        # Run immediately, and then every 30 seconds:
        self._lc.start(30, now=True)

    def _download(self):
        """Download the page."""
        print("Downloading!")
        def parse(result):
            print("Got %r back from Yahoo." % (result,))
            values = result.strip().split(",")
            self._value = float(values[1])
        d = getPage(
            "http://download.finance.yahoo.com/d/quotes.csv?e=.csv&f=c4l1&s=%s=X"
            % (self._name,))
        d.addCallback(parse)
        d.addErrback(log.err)
        return d


# Blocking wrapper:
class ExchangeRate(object):
    """Blocking API for downloading exchange rate."""

    def __init__(self, name):
        self._exchange = _ExchangeRate(name)
```

(continues on next page)

(continued from previous page)

```
    @run_in_reactor
    def start(self):
        self._exchange.start()

    @wait_for(timeout=1)
    def latest_value(self):
        """Return the latest exchange rate value.

        May be None if no value is available.
        """
        return self._exchange.latest_value()


EURUSD = ExchangeRate("EURUSD")
app = Flask(__name__)

@app.route('/')
def index():
    rate = EURUSD.latest_value()
    if rate is None:
        rate = "unavailable, please refresh the page"
    return "Current EUR/USD exchange rate is %s." % (rate,)


if __name__ == '__main__':
    import sys, logging
    logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
    EURUSD.start()
    app.run()
```

1.3.2 Minimize decorated code

It's best to have as little code as possible in the `@wait_for`/`@run_in_reactor` wrappers. As this code straddles two worlds (or at least, two threads) it is more difficult to unit test. Having an extra layer between this code and the application code is also useful in this regard as well: Twisted code can be pushed into the lower-level Twisted layer, and code hiding the Twisted details from the application code can be pushed into the higher-level layer.

1.4 Static Type Checking

Crochet comes with type hints for Python 3.6+. However, due to current limitations in `Callable` generic construction (see PEP 612 — Parameter Specification Variables), the arguments of a call to a `@run_in_reactor`-decorated function or method cannot be checked without giving type checkers some special help. Crochet ships with a plugin which fills this role when the `mypy` static type checker is used. It resides in `crochet.mypy` and must be configured as described in Configuring mypy to use plugins. For example, in a `mypy.ini` configuration file:

```
[mypy]
plugins = crochet.mypy
```

This type checking is intended primarily for code which calls the decorated function. As Twisted isn't fully type-hinted yet, and in particular Deferred does not yet have a generic type argument so that the eventual result type can vary, the analysis of the return type of a `@run_in_reactor` function/method does not account for a Deferred result. This

requires you to lie to the type checker when returning a Deferred; just cast it to the known, eventual result type using `typing.cast`. For example:

```
@run_in_reactor
def get_time_in_x_seconds(delay: float) -> float:
    def get_time() -> float:
        return reactor.seconds()  # type: ignore

    if delay < 0.001:
        # Close enough; just return the current time.
        return get_time()
    else:
        d = Deferred()

        def complete():
            d.callback(get_time())

        reactor.callLater(delay, complete)  # type: ignore
        return typing.cast(float, d)
```

If the mypy plugin is correctly installed, the client code will expect a float from the `wait()` of the `EventualResult` returned by a call to this function:

```
# OK
t1: float = get_time_in_x_seconds(2).wait(3)
print(f"The reactor time is {t1}")

# mypy error: Incompatible types in assignment
#   (expression has type "float", variable has type "str")
t2: str = get_time_in_x_seconds(2).wait(3)
print(f"The reactor time is {t2}")
```

1.5 Known Issues and Workarounds

1.5.1 Don't Call Twisted APIs from non-Twisted threads

As is the case in any Twisted program, you should never call Twisted APIs (e.g. `reactor.callLater`) from non-Twisted threads. Only call Twisted APIs from functions decorated by `@wait_for` and friends.

1.5.2 Preventing deadlocks on shutdown

To ensure a timely process exit, during reactor shutdown Crochet will try to interrupt calls to `EventualResult.wait()` or functions decorated with `@wait_for` with a `crochet.ReactorStopped` exception. This is still not a complete solution, unfortunately. If you are shutting down a thread pool as part of Twisted's reactor shutdown, this will wait until all threads are done. If you're blocking indefinitely, this may rely on Crochet interrupting those blocking calls... but Crochet's shutdown may be delayed until the thread pool finishes shutting down, depending on the ordering of shutdown events.

The solution is to interrupt all blocking calls yourself. You can do this by firing or canceling any `Deferred` instances you are waiting on as part of your application shutdown, and do so before you stop any thread pools.

1.5.3 Reducing Twisted log messages

Twisted can be rather verbose with its log messages. If you wish to reduce the message flow you can limit them to error messages only:

```
import logging
logging.getLogger('twisted').setLevel(logging.ERROR)
```

1.5.4 Missing tracebacks

In order to prevent massive memory leaks, Twisted currently wipes out the traceback from exceptions it captures (see https://tm.tl/7873 for ideas on improving this). This means that often exceptions re-raised by Crochet will be missing their tracebacks. You can however get access to a string version of the traceback, suitable for logging, from `EventualResult` objects returned by `@run_in_reactor`-wrapped functions:

```
from crochet import run_in_reactor, TimeoutError

@run_in_reactor
def download_page(url):
    from twisted.web.client import getPage
    return getPage(url)

result = download_page("https://github.com")
try:
    page = result.wait(timeout=1000)
except TimeoutError:
    # Handle timeout ...
except:
    # Something else happened:
    print(result.original_failure().getTraceback())
```

1.5.5 uWSGI, multiprocessing, Celery

uWSGI, the standard library `multiprocessing.py` library and Celery by default use `fork()` without `exec()` to create child processes on Unix systems. This means they effectively clone a running parent Python process, preserving all existing imported modules. This is a fundamentally broken thing to do, e.g. it breaks the standard library's `logging` package. It also breaks Crochet.

You have two options for dealing with this problem. The ideal solution is to avoid this "feature":

uWSGI Use the `--lazy-apps` command-line option.

`multiprocessing.py` Use the `spawn` (or possibly `forkserver`) start methods when using Python 3. See https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods for more details.

Alternatively, you can ensure you only start Crochet inside the child process:

uWSGI Only run `crochet.setup()` inside the WSGI application function.

`multiprocessing.py` Only run `crochet.setup()` in the child process.

Celery Only run `crochet.setup()` inside tasks.

1.6 Differences from async/await

Python 3.6 introduces a new mechanism, `async`/`await`, that allows integrating asynchronous code in a seemingly blocking way. This mechanism is however quite different than Crochet.

`await` gives the illusion of blocking, but can only be used in functions that are marked as `async`. As such, this is not true blocking integration: the asyncess percolates throughout your program and cannot be restricted to just a single function.

In contrast, Crochet allows you to truly block on an asynchronous event: it's just another blocking function call, and can be used in any normal Python function.

1.7 API Reference

crochet.**setup**()
Initialize the crochet library.

This starts the reactor in a thread, and connect's Twisted's logs to Python's standard library logging module.

This must be called at least once before the library can be used, and can be called multiple times.

crochet.**no_setup**()
Initialize the crochet library with no side effects.

No reactor will be started, logging is uneffected, etc.. Future calls to setup() will have no effect. This is useful for applications that intend to run Twisted's reactor themselves, and so do not want libraries using crochet to attempt to start it on their own.

If no_setup() is called after setup(), a RuntimeError is raised.

crochet.**run_in_reactor**(*function*)
A decorator that ensures the wrapped function runs in the reactor thread.

When the wrapped function is called, an EventualResult is returned.

crochet.**wait_for**(*timeout*)
A decorator factory that ensures the wrapped function runs in the reactor thread.

When the wrapped function is called, its result is returned or its exception raised. Deferreds are handled transparently. Calls will timeout after the given number of seconds (a float), raising a crochet.TimeoutError, and cancelling the Deferred being waited on.

class crochet.**EventualResult**(*deferred*, *_reactor*)
A blocking interface to Deferred results.

This allows you to access results from Twisted operations that may not be available immediately, using the wait() method.

In general you should not create these directly; instead use functions decorated with @run_in_reactor.

cancel()
Try to cancel the operation by cancelling the underlying Deferred.

Cancellation of the operation may or may not happen depending on underlying cancellation support and whether the operation has already finished. In any case, however, the underlying Deferred will be fired.

Multiple calls will have no additional effect.

original_failure()
Return the underlying Failure object, if the result is an error.

If no result is yet available, or the result was not an error, None is returned.

This method is useful if you want to get the original traceback for an error result.

stash()
Store the EventualResult in memory for later retrieval.

Returns a integer uid which can be passed to crochet.retrieve_result() to retrieve the instance later on.

wait(*timeout*)
Return the result, or throw the exception if result is a failure.

It may take an unknown amount of time to return the result, so a timeout option is provided. If the given number of seconds pass with no result, a TimeoutError will be thrown.

If a previous call timed out, additional calls to this function will still wait for a result and return it if available. If a result was returned or raised on one call, additional calls will return/raise the same result.

crochet.**retrieve_result**(*result_id*)
Return the given EventualResult, and remove it from the store.

exception crochet.**TimeoutError**
A timeout has been hit.

exception crochet.**ReactorStopped**
The reactor has stopped, and therefore no result will ever become available from this EventualResult.

1.8 What's New

1.8.1 2.1.0

- Various internal modernizations and maintenance.
- Dropped Python 3.6 and 3.7 support.

1.8.2 2.0.0

New features:

- It's possible to decorate `async/await` Twisted functions with `@wait_for` and `@run_in_reactor`, thanks to Árni Már Jónsson.
- Added type hints, thanks to Merlin Davis.
- Added formal support for Python 3.9.

Removed features:

- Dropped the deprecated APIs `@wait_for_reactor`, `@in_reactor`, `DeferredResult`, the `wrapped_function` attribute, and unlimited timeouts on `EventualResult.wait()`.
- Dropped support for Python 2.7 and 3.5.

1.8.3 1.12.0

Bug fixes:

- Fix a timeout overflow bug in 32-bit machines.

1.8.4 1.11.0

New features:

- Added support for Python 3.8 and PyPy 3.

Backwards incompatibility:

- Dropped support for Python 3.4, since latest Twisted doesn't support it.

1.8.5 1.10.0

New features:

- Added support for Python 3.7. Thanks to Jeremy Cline for the patch.

1.8.6 1.9.0

New features:

- The underlying callable wrapped `@run_in_reactor` and `@wait_for` is now available via the more standard `__wrapped__` attribute.

Backwards incompatibility (in tests):

- This was actually introduced in 1.8.0: `wrapped_function` may not always be available on decorated callables. You should use `__wrapped__` instead.

Bug fixes:

- Fixed regression in 1.8.0 where bound method couldn't be wrapped. Thanks to 2mf for the bug report.

1.8.7 1.8.0

New features:

- Signatures on decorated functions now match the original functions. Thanks to Mikhail Terekhov for the original patch.
- Documentation improvements, including an API reference.

Bug fixes:

- Switched to EPoll reactor for logging thread. Anecdotal evidence suggests this fixes some issues on AWS Lambda, but it's not clear why. Thanks to Rolando Espinoza for the patch.
- It's now possible to call `@run_in_reactor` and `@wait_for` above a `@classmethod`. Thanks to vak for the bug report.

1.8.8 1.7.0

Bug fixes:

- If the Python `logging.Handler` throws an exception Crochet no longer goes into a death spiral. Thanks to Michael Schlenker for the bug report.

Removed features:

- Versions of Twisted < 16.0 are no longer supported (i.e. no longer tested in CI.)

1.8.9 1.6.0

New features:

- Added support for Python 3.6.

1.8.10 1.5.0

New features:

- Added support for Python 3.5.

Removed features:

- Python 2.6, Python 3.3, and versions of Twisted < 15.0 are no longer supported.

1.8.11 1.4.0

New features:

- Added support for Python 3.4.

Documentation:

- Added a section on known issues and workarounds.

Bug fixes:

- Main thread detection (used to determine when Crochet should shutdown) is now less fragile. This means Crochet now supports more environments, e.g. uWSGI. Thanks to Ben Picolo for the patch.

1.8.12 1.3.0

Bug fixes:

- It is now possible to call `EventualResult.wait()` (or functions wrapped in `wait_for`) at import time if another thread holds the import lock. Thanks to Ken Struys for the patch.

1.8.13 1.2.0

New features:

- `crochet.wait_for` implements the timeout/cancellation pattern documented in previous versions of Crochet. `crochet.wait_for_reactor` and `EventualResult.wait(timeout=None)` are now deprecated, since lacking timeouts they could potentially block forever.
- Functions wrapped with `wait_for` and `run_in_reactor` can now be accessed via the `wrapped_function` attribute, to ease unit testing of the underlying Twisted code.

API changes:

- It is no longer possible to call `EventualResult.wait()` (or functions wrapped with `wait_for`) at import time, since this can lead to deadlocks or prevent other threads from importing. Thanks to Tom Prince for the bug report.

Bug fixes:

- `warnings` are no longer erroneously turned into Twisted log messages.

- The reactor is now only imported when `crochet.setup()` or `crochet.no_setup()` are called, allowing daemonization if only `crochet` is imported (http://tm.tl/7105). Thanks to Daniel Nephin for the bug report.

Documentation:

- Improved motivation, added contact info and news to the documentation.
- Better example of using Crochet from a normal Twisted application.

1.8.14 1.1.0

Bug fixes:

- `EventualResult.wait()` can now be used safely from multiple threads, thanks to Gavin Panella for reporting the bug.
- Fixed reentrancy deadlock in the logging code caused by http://bugs.python.org/issue14976, thanks to Rod Morehead for reporting the bug.
- Crochet now installs on Python 3.3 again, thanks to Ben Cordero.
- Crochet should now work on Windows, thanks to Konstantinos Koukopoulos.
- Crochet tests can now run without adding its absolute path to PYTHONPATH or installing it first.

Documentation:

- `EventualResult.original_failure` is now documented.

1.8.15 1.0.0

Documentation:

- Added section on use cases and alternatives. Thanks to Tobias Oberstein for the suggestion.

Bug fixes:

- Twisted does not have to be pre-installed to run `setup.py`, thanks to Paul Weaver for bug report and Chris Scutcher for patch.
- Importing Crochet does not have side-effects (installing reactor event) any more.
- Blocking calls are interrupted earlier in the shutdown process, to reduce scope for deadlocks. Thanks to rmorehead for bug report.

1.8.16 0.9.0

New features:

- Expanded and much improved documentation, including a new section with design suggestions.
- New decorator `@wait_for_reactor` added, a simpler alternative to `@run_in_reactor`.
- Refactored `@run_in_reactor`, making it a bit more responsive.
- Blocking operations which would otherwise never finish due to reactor having stopped (`EventualResult.wait()` or `@wait_for_reactor` decorated call) will be interrupted with a `ReactorStopped` exception. Thanks to rmorehead for the bug report.

Bug fixes:

- `@run_in_reactor` decorated functions (or rather, their generated wrapper) are interrupted by Ctrl-C.
- On POSIX platforms, a workaround is installed to ensure processes started by *reactor.spawnProcess* have their exit noticed. See Twisted ticket 6378 for more details about the underlying issue.

1.8.17 0.8.1

- `EventualResult.wait()` now raises error if called in the reactor thread, thanks to David Buchmann.
- Unittests are now included in the release tarball.
- Allow Ctrl-C to interrupt `EventualResult.wait(timeout=None)`.

1.8.18 0.7.0

- Improved documentation.

1.8.19 0.6.0

- Renamed `DeferredResult` to `EventualResult`, to reduce confusion with Twisted's `Deferred` class. The old name still works, but is deprecated.
- Deprecated `@in_reactor`, replaced with `@run_in_reactor` which doesn't change the arguments to the wrapped function. The deprecated API still works, however.
- Unhandled exceptions in `EventualResult` objects are logged.
- Added more examples.
- `setup.py sdist` should work now.

1.8.20 0.5.0

- Initial release.

Index

C

cancel() (*crochet.EventualResult method*), 19

E

EventualResult (*class in crochet*), 19

N

no_setup() (*in module crochet*), 19

O

original_failure() (*crochet.EventualResult method*), 19

R

ReactorStopped, 20
retrieve_result() (*in module crochet*), 20
run_in_reactor() (*in module crochet*), 19

S

setup() (*in module crochet*), 19
stash() (*crochet.EventualResult method*), 20

T

TimeoutError, 20

W

wait() (*crochet.EventualResult method*), 20
wait_for() (*in module crochet*), 19