

```

import asyncio
import time
import logging
from itertools import chain

from elasticsearch import Transport, TransportError, ConnectionTimeout, ConnectionError, SerializationError

from .connection import AIOHttpConnection
from .helpers import ensure_future

logger = logging.getLogger('elasticsearch')

class AsyncTransport(Transport):
    def __init__(self, hosts, connection_class=AIOHttpConnection, loop=None, sniff_on_start=False, raise_on_sniff_error=True, **kwargs):
        self.raise_on_sniff_error = raise_on_sniff_error
        self.loop = asyncio.get_event_loop() if loop is None else loop
        kwargs['loop'] = self.loop
        super().__init__(hosts, connection_class=connection_class, sniff_on_start=False, **kwargs)

        self.sniffing_task = None
        if sniff_on_start:
            # schedule sniff on start
            self.initiate_sniff(True)

    def initiate_sniff(self, initial=False):
        """
        Initiate a sniffing task. Make sure we only have one sniff request
        running at any given time. If a finished sniffing request is around,
        collect its result (which can raise its exception).
        """
        if self.sniffing_task and self.sniffing_task.done():
            try:
                if self.sniffing_task is not None:
                    self.sniffing_task.result()
            except:
                if self.raise_on_sniff_error:
                    raise
            finally:
                self.sniffing_task = None

        if self.sniffing_task is None:
            self.sniffing_task = ensure_future(self.sniff_hosts(initial), loop=self.loop)

    def close(self):
        if self.sniffing_task:
            self.sniffing_task.cancel()
        super().close()

    def get_connection(self):
        if self.sniffer_timeout:
            if time.time() >= self.last_sniff + self.sniffer_timeout:
                self.initiate_sniff()
        return self.connection_pool.get_connection()

    def mark_dead(self, connection):
        self.connection_pool.mark_dead(connection)

```

```

    if self.sniff_on_connection_fail:
        self.initiate_sniff()

@asyncio.coroutine
def _get_sniff_data(self, initial=False):
    previous_sniff = self.last_sniff

    # reset last_sniff timestamp
    self.last_sniff = time.time()

    # use small timeout for the sniffing request, should be a fast api call
    timeout = self.sniff_timeout if not initial else None

    tasks = [
        c.perform_request('GET', '/_nodes/_all/clear', timeout=timeout)
        # go through all current connections as well as the
        # seed_connections for good measure
        for c in chain(self.connection_pool.connections, (c for c in self.se
ed_connections if c not in self.connection_pool.connections))
    ]

    done = ()
    try:
        while tasks:
            # execute sniff requests in parallel, wait for first to return
            done, tasks = yield from asyncio.wait(tasks, return_when=asyncio
.FIRST_COMPLETED, loop=self.loop)
            # go through all the finished tasks
            for t in done:
                try:
                    _, headers, node_info = t.result()
                    node_info = self.deserializer.loads(node_info, headers.g
et('content-type'))
                except (ConnectionError, SerializationError) as e:
                    logger.warn('Sniffing request failed with %r', e)
                    continue
                node_info = list(node_info['nodes'].values())
                return node_info
            else:
                # no task has finished completely
                raise TransportError("N/A", "Unable to sniff hosts.")
    except:
        # keep the previous value on error
        self.last_sniff = previous_sniff
        raise
    finally:
        # clean up pending futures
        for t in chain(done, tasks):
            t.cancel()

@asyncio.coroutine
def sniff_hosts(self, initial=False):
    """

```

Obtain a list of nodes from the cluster and create a new connection pool using the information retrieved.

To extract the node connection parameters use the “nodes_to_host_callback”.

:arg initial: flag indicating if this is during startup
 (“sniff_on_start”), ignore the “sniff_timeout” if “True”

```

"""
    node_info = yield from self._get_sniff_data(initial)

    hosts = list(filter(None, (self._get_host_info(n) for n in node_info)))

    # we weren't able to get any nodes, maybe using an incompatible
    # transport_schema or host_info_callback blocked all - raise error.
    if not hosts:
        raise TransportError("N/A", "Unable to sniff hosts - no viable hosts found.")

    # remember current live connections
    orig_connections = self.connection_pool.connections[:]
    self.set_connections(hosts)
    # close those connections that are not in use any more
    for c in orig_connections:
        if c not in self.connection_pool.connections:
            yield from c.close()

@asyncio.coroutine
def main_loop(self, method, url, params, body, ignore=(), timeout=None):
    for attempt in range(self.max_retries + 1):
        connection = self.get_connection()

        try:
            status, headers, data = yield from connection.perform_request(
                method, url, params, body, ignore=ignore, timeout=timeout)

        except TransportError as e:
            if method == 'HEAD' and e.status_code == 404:
                return False

            retry = False
            if isinstance(e, ConnectionTimeout):
                retry = self.retry_on_timeout
            elif isinstance(e, ConnectionError):
                retry = True
            elif e.status_code in self.retry_on_status:
                retry = True

            if retry:
                # only mark as dead if we are retrying
                self.mark_dead(connection)
                # raise exception on last retry
                if attempt == self.max_retries:
                    raise
            else:
                raise

        else:
            if method == 'HEAD':
                return 200 <= status < 300

            # connection didn't fail, confirm it's live status
            self.connection_pool.mark_live(connection)
            if data:
                data = self.deserializer.loads(data, headers.get('content-type'))

            return data

    def perform_request(self, method, url, params=None, body=None):

```

```
if body is not None:
    body = self.serializer.dumps(body)

    # some clients or environments don't support sending GET with body
    if method in ('HEAD', 'GET') and self.send_get_body_as != 'GET':
        # send it as post instead
        if self.send_get_body_as == 'POST':
            method = 'POST'

        # or as source parameter
        elif self.send_get_body_as == 'source':
            if params is None:
                params = {}
            params['source'] = body
            body = None

if body is not None:
    try:
        body = body.encode('utf-8')
    except (UnicodeDecodeError, AttributeError):
        # bytes/str - no need to re-encode
        pass

ignore = ()
timeout = None
if params:
    timeout = params.pop('request_timeout', None)
    ignore = params.pop('ignore', ())
    if isinstance(ignore, int):
        ignore = (ignore, )

return ensure_future(self.main_loop(method, url, params, body,
                                     ignore=ignore,
                                     timeout=timeout),
                    loop=self.loop)
```