getdns Documentation

Release v1.0.0b1

Melinda Shore, Gowri Visweswaran

Nov 22, 2017

Contents

1 Dependencies 3

2 Building 5

3 Using getdns 7

4 Module-level attributes and methods 9

5 Known issues 11

6 Indices and tables 25

Python Module Index 27

"getdns" is an implementation of Python language bindings for the getdns API. getdns is a modern, asynchronous DNS API that simplifies access to advanced DNS features, including DNSSEC. The API specification was developed by Paul Hoffman. getdns is built on top of the getdns implementation developed as a joint project between Verisign Labs and NLnet Labs.

We have tried to keep this interface as Pythonic as we can while staying true to the getdns architecture, including trying to maintain consistency with Python object design.

CHAPTER 1

Dependencies

This version of getdns has been built and tested against Python 2.7 and Python 3.4. We also expect these other prerequisites to be installed:

- libgetdns, version 0.9.0 or later
- libunbound, version 1.4.16 or later
- libidn version 1

This release has been tested against libgetdns 0.9.0.

CHAPTER 2

Building

The code repository for getdns is available at: https://github.com/getdnsapi/getdns-python-bindings. If you are building from source you will need the Python development package for Python 2.7. On Linux systems this is typically something along the lines of "python-dev" or "python2.7-dev", available through your package system. On Mac OS we are building against the python.org release, available in source form here.

For the actual build, we are using the standard Python distutils. To build and install:

```
python setup.py build
python setup.py install
```

If you have installed getdns libraries and headers in other than the default location, build the Python bindings using the `--with-getdns` argument to setup.py, providing the getdns root directory as an argument. (Note that there should be a space between –with-getdns and the directory). For example,

```
python setup.py build --with-getdns ~/build
```

if you installed getdns into your home/build directory.

We've added optional support for draft-ietf-dnsop-cookies. It is implemented as a getdns extension (see below). It is not built by default. To enable it, you must build libgetdns with cookies support and add the `--with-edns-cookies` to the Python module build (i.e. `python setup.py build --with-edns-cookies`).

CHAPTER 3

Using getdns

3.1 Contexts

All getdns queries happen within a resolution *context*, and among the first tasks you'll need to do before issuing a query is to acquire a Context object. A context is an opaque object with attributes describing the environment within which the query and replies will take place, including elements such as DNSSEC validation, whether the resolution should be performed as a recursive resolver or a stub resolver, and so on. Individual Context attributes may be examined directly, and the overall state of a given context can be queried with the Context.get_api_information() method.

See section 8 of the API specification

3.2 Examples

In this example, we do a simple address lookup and dump the results to the screen:

```
import getdns, pprint, sys

def main():
    if len(sys.argv) != 2:
        print "Usage: {0} hostname".format(sys.argv[0])
        sys.exit(1)

    ctx = getdns.Context()
    extensions = { "return_both_v4_and_v6" :
    getdns.EXTENSION_TRUE }
    results = ctx.address(name=sys.argv[1],
    extensions=extensions)
    if results.status == getdns.RESPSTATUS_GOOD:
        sys.stdout.write("Addresses: ")

        for addr in results.just_address_answers:
            print " {0}".format(addr["address_data"])
        sys.stdout.write("\n\n")
        print "Entire results tree: "
        pprint.pprint(results.replies_tree)
    if results.status == getdns.RESPSTATUS_NO_NAME:
        print "{0} not found".format(sys.argv[1])

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

In this example, we do a DNSSEC query and check the response:

```
import getdns, sys

dnssec_status = {
    "DNSSEC_SECURE" : 400,
    "DNSSEC_BOGUS" : 401,
    "DNSSEC_INDETERINATE" : 402,
    "DNSSEC_INSECURE" : 403,
    "DNSSEC_NOT_PERFORMED" : 404
}


def dnssec_message(value):
    for message in dnssec_status.keys():
        if dnssec_status[message] == value:
            return message

def main():
    if len(sys.argv) != 2:
        print "Usage: {0} hostname".format(sys.argv[0])
        sys.exit(1)

    ctx = getdns.Context()
    extensions = { "return_both_v4_and_v6" :
    getdns.EXTENSION_TRUE,
                   "dnssec_return_status" :
                   getdns.EXTENSION_TRUE }
    results = ctx.address(name=sys.argv[1],
    extensions=extensions)
    if results.status == getdns.RESPSTATUS_GOOD:
        sys.stdout.write("Addresses: ")
        for addr in results.just_address_answers:
            print " {0}".format(addr["address_data"])
        sys.stdout.write("\n")

        for result in results.replies_tree:
            if "dnssec_status" in result.keys():
                print "{0}: dnssec_status:
                {1}".format(result["canonical_name"],
                                                       dnssec_message(result["dnssec_
↪status"]))

    if results.status == getdns.RESPSTATUS_NO_NAME:
        print "{0} not found".format(sys.argv[1])


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

CHAPTER 4

Module-level attributes and methods

`__version__`

The `getdns.__version__` attribute contains the version string for the Python getdns module. Please note that this is independent of the version of the underlying getdns library, which may be retrieved through attributes associated with a Context.

`get_errorstr_by_id``()`

Returns a human-friendly string representation of an error ID.

`ulabel_to_alabel``()`

Converts a ulabel to an alabel. Takes one argument (the ulabel)

`alabel_to_ulabel``()`

Converts an alabel to a ulabel. Takes one argument (the alabel)

`root_trust_anchor``()`

Returns the default root trust anchor for DNSSEC.

CHAPTER 5

Known issues

- "userarg" currently only accepts a string. This will be changed in a future release, to take arbitrary data types

Contents:

5.1 `getdns` reference

5.1.1 getdns contexts

This section describes the *getdns* Context object, as well as its as its methods and attributes.

class `getdns.`**`Context`**`(`[*set_from_os*]`)`

Creates a *context*, an opaque object which describes the environment within which a DNS query executes. This includes namespaces, root servers, resolution types, and so on. These are accessed programmatically through the attributes described below.

Context() takes one optional constructor argument. `set_from_os` is an integer and may take the value either 0 or 1. If 1, which most developers will want, getdns will populate the context with default values for the platform on which it's running.

The `Context` class has the following public read/write attributes:

`append_name`

Specifies whether to append a suffix to the query string before the API starts resolving a name. Its value must be one of `getdns.APPEND_NAME_ALWAYS`, `getdns.APPEND_NAME_ONLY_TO_SINGLE_LABEL_AFTER_FAILURE`, `getdns.APPEND_NAME_ONLY_TO_MULTIPLE_LABEL_NAME_AFTER_FAILURE`, or `getdns.APPEND_NAME_NEVER`. This controls whether or not to append the suffix given by `suffix`.

`dns_root_servers`

The value of *dns_root_servers* is a list of dictionaries containing addresses to be used for looking up top-level domains. Each dict in the list contains two key-value pairs:

- address_data: a string representation of an IPv4 or IPv6 address

- address_type: either the string "IPv4" or "IPv6"

For example, the addresses list could look like

```
>>> addrs = [ { 'address_data': '2001:7b8:206:1::4:53',
    'address_type': 'IPv6' },
...         { 'address_data': '65.22.9.1',
    'address_type': 'IPv4' } ]
>>> mycontext.dns_root_servers = addrs
```

`dns_transport_list`
An ordered list of transport options to be used for DNS lookups, ordered by preference (first choice as list element 0, second as list element 1, and so on). The possible values are `getdns.TRANSPORT_UDP`, `getdns.TRANSPORT_TCP`, and `getdns.TRANSPORT_TLS`.

`dnssec_allowed_skew`
Its value is the number of seconds of skew that is allowed in either direction when checking an RRSIG's Expiration and Inception fields. The default is 0.

`dnssec_trust_anchors`
Its value is a list of DNSSEC trust anchors, expressed as RDATAs from DNSKEY resource records.

`edns_client_subnet_private`
May be set to 0 or 1. When 1, requests upstreams not to reveal query's originating network.

`edns_do_bit`
Its value must be an integer valued either 0 or 1. The default is 0.

`edns_extended_rcode`
Its value must be an integer between 0 and 255, inclusive. The default is 0.

`edns_maximum_udp_payload_size`
Its value must be an integer between 512 and 65535, inclusive. The default is 512.

`edns_version`
Its value must be an integer between 0 and 255, inclusive. The default is 0.

`follow_redirects`
Specifies whether or not DNS queries follow redirects. The value must be one of `getdns.REDIRECTS_FOLLOW` for normal following of redirects though CNAME and DNAME; or `getdns.REDIRECTS_DO_NOT_FOLLOW` to cause any lookups that would have gone through CNAME and DNAME to return the CNAME or DNAME, not the eventual target.

`idle_timeout`
The idle timeout for TCP connections.

`implementation_string`
A string describing the implementation of the underlying getdns library, retrieved from libgetdns. Currently "https://getdnsapi.net"

`limit_outstanding_queries`
Specifies *limit* (an integer value) on the number of outstanding DNS queries. The API will block itself from sending more queries if it is about to exceed this value, and instead keep those queries in an internal queue. The a value of 0 indicates that the number of outstanding DNS queries is unlimited.

`namespaces`
The *namespaces* attribute takes an ordered list of namespaces that will be queried. (*Important: this context setting is ignored for the getdns.general() function; it is used for the other functions.*) The allowed values are `getdns.NAMESPACE_DNS`, `getdns.NAMESPACE_LOCALNAMES`, `getdns.NAMESPACE_NETBIOS`, `getdns.NAMESPACE_MDNS`, and `getdns.NAMESPACE_NIS`. When a

normal lookup is done, the API does the lookups in the order given and stops when it gets the first result; a different method with the same result would be to run the queries in parallel and return when it gets the first result. Because lookups might be done over different mechanisms because of the different namespaces, there can be information leakage that is similar to that seen with POSIX *getaddrinfo()*. The default is determined by the OS.

`resolution_type`
Specifies whether DNS queries are performed with nonrecursive lookups or as a stub resolver. The value is either `getdns.RESOLUTION_RECURSING` or `getdns.RESOLUTION_STUB`.

If an implementation of this API is only able to act as a recursive resolver, setting *resolution_type* to `getdns.RESOLUTION_STUB` will throw an exception.

`suffix`
Its value is a list of strings to be appended based on *`append_name`*. The list elements must follow the rules in **RFC 4343#section-2.1**

`timeout`
Its value must be an integer specifying a timeout for a query, expressed in milliseconds.

`tls_authentication`
The mechanism to be used for authenticating the TLS server when using a TLS transport. May be `getdns.AUTHENTICATION_REQUIRED` or `getdns.AUTHENTICATION_NONE`. (getdns.AUTHENTICATION_HOSTNAME remains as an alias for getdns.AUTHENTICATION_REQUIRED but is deprecated and will be removed in a future release)

`tls_query_padding_blocksize`
Optional padding blocksize for queries when using TLS. Used to increase the difficulty for observers to guess traffic content.

`upstream_recursive_servers`
A list of dicts defining where a stub resolver will send queries. Each dict in the list contains at least two names: address_type (either "IPv4" or "IPv6") and address_data (whose value is a string representation of an IP address). It might also contain "port" to specify which port to use to contact these DNS servers; the default is 53. If the stub and a recursive resolver both support TSIG (RFC 2845), the upstream_list entry can also contain tsig_algorithm (a string) that is the name of the TSIG hash algorithm, and tsig_secret (a base64 string) that is the TSIG key.

There is also now support for pinning an upstream's certificate's public keys with pinsets (when using TLS for transport). Add an element to the upstream_recursive_server list entry, called 'tls_pubkey_pinset', which is a list of public key pins. (See the example code in our examples directory).

`version_string`
The libgetdns version, retrieved from the underlying getdns library.

The *`Context`* class includes public methods to execute a DNS query, as well as a method to return the entire set of context attributes as a Python dictionary. *`Context`* methods are described below:

`general` (*name*, *request_type*[, *extensions*][, *userarg*][, *transaction_id*][, *callback*])
`Context.general()` is used for looking up any type of DNS record. The keyword arguments are:

- `name`: a string containing the query term.
- `request_type`: a DNS RR type as a getdns constant (listed here)
- `extensions`: optional. A dictionary containing attribute/value pairs, as described below
- `userarg`: optional. A string containing arbitrary user data; this is opaque to getdns
- `transaction_id`: optional. An integer.

- `callback`: optional. This is a function name. If it is present the query will be performed asynchronously (described below).

address (*name*[, *extensions*][, *userarg*][, *transaction_id*][, *callback*])
There are two critical differences between `Context.address()` and `Context.general()` beyond the missing *request_type* argument:

- In `Context.address()`, the name argument can only take a host name.
- `Context.address()` always uses all of namespaces from the context (to better emulate getaddrinfo()), while `Context.general()` only uses the DNS namespace.

hostname (*name*[, *extensions*][, *userarg*][, *transaction_id*][, *callback*])
The address is given as a dictionary. The dictionary must have two names:

- `address_type`: must be a string matching either "IPv4" or "IPv6"
- `address_data`: a string representation of an IPv4 or IPv6 IP address

service (*name*[, *extensions*][, *userarg*][, *transaction_id*][, *callback*])
`name` must be a domain name for an SRV lookup. The call returns the relevant SRV information for the name

get_api_information ()
Retrieves context information. The information is returned as a Python dictionary with the following keys:

- `version_string`
- `implementation_string`
- `resolution_type`
- `all_context`

`all_context` is a dictionary containing the following keys:

- `append_name`
- `dns_transport`
- `dnssec_allowed_skew`
- `edns_do_bit`
- `edns_extended_rcode`
- `edns_version`
- `follow_redirects`
- `limit_outstanding_queries`
- `namespaces`
- `suffix`
- `timeout`
- `tls_authentication`
- `upstream_recursive_servers`

get_supported_attributes ()
Returns a list of the attributes supported by this Context object.

The `getdns` module has the following read-only attribute:

getdns.**__version__**

Specifies the version string for the getdns python module

5.1.2 Extensions

Extensions are Python dictionaries, with the keys being the names of the extensions. The definition of each extension describes the values that may be assigned to that extension. For most extensions it is a Boolean, and since the default value is "False" it will most often take the value `getdns.EXTENSION_TRUE`.

The extensions currently supported by *getdns* are:

- `dnssec_return_status`
- `dnssec_return_only_secure`
- `dnssec_return_validation_chain`
- `return_both_v4_and_v6`
- `add_opt_parameters`
- `add_warning_for_bad_dns`
- `specify_class`
- `return_call_reporting`

Extensions for DNSSEC

If an application wants the API to do DNSSEC validation for a request, it must set one or more DNSSEC-related extensions. Note that the default is for none of these extensions to be set and the API will not perform DNSSEC validation. Note that getting DNSSEC results can take longer in a few circumstances.

To return the DNSSEC status for each DNS record in the `replies_tree` list, use the `dnssec_return_status` extension. Set the extension's value to `getdns.EXTENSION_TRUE` to cause the returned status to have the name `dnssec_status` added to the other names in the record's dictionary ("header", "question", and so on). The potential values for that name are `getdns.DNSSEC_SECURE`, `getdns.DNSSEC_BOGUS`, `getdns.DNSSEC_INDETERMINATE`, and `getdns.DNSSEC_INSECURE`.

If instead of returning the status, you want to only see secure results, use the `dnssec_return_only_secure` extension. The extension's value is set to `getdns.EXTENSION_TRUE` to cause only records that the API can validate as secure with DNSSEC to be returned in the `replies_tree` and `replies_full lists`. No additional names are added to the dict of the record; the change is that some records might not appear in the results. When this context option is set, if the API receives DNS replies but none are determined to be secure, the error code at the top level of the `response` object is `getdns.RESPSTATUS_NO_SECURE_ANSWERS`.

Applications that want to do their own validation will want to have the DNSSEC-related records for a particular response. Use the `dnssec_return_validation_chain` extension. Set the extension's value to `getdns.EXTENSION_TRUE` to cause a set of additional DNSSEC-related records needed for validation to be returned in the `response object`. This set comes as `validation_chain` (a list) at the top level of the `response` object. This list includes all resource record dicts for all the resource records (DS, DNSKEY and their RRSIGs) that are needed to perform the validation from the root up.

If a request is using a context in which stub resolution is set, and that request also has any of the `dnssec_return_status`, `dnssec_return_only_secure`, or `dnssec_return_validation_chain` extensions specified, the API will not perform the request and will instead return an error of `getdns.RETURN_DNSSEC_WITH_STUB_DISALLOWED`.

Returning both IPv4 and IPv6 responses

Many applications want to get both IPv4 and IPv6 addresses in a single call so that the results can be processed together. The `address()` method is able to do this automatically. If you are using the `general()` method, you can enable this with the `return_both_v4_and_v6` extension. The extension's value must be set to `getdns.EXTENSION_TRUE` to cause the results to be the lookup of either A or AAAA records to include any A and AAAA records for the queried name (otherwise, the extension does nothing). These results are expected to be usable with Happy Eyeballs systems that will find the best socket for an application.

Setting up OPT resource records

For lookups that need an **OPT** resource record in the Additional Data section, use the `add_opt_parameters` extension. The extension's value (a dict) contains the parameters; these are described in more detail in **RFC 2671**. They are:

- `maximum_udp_payload_size`: an integer between 512 and 65535 inclusive. If not specified it defaults to the value in the getdns context.
- `extended_rcode`: an integer between 0 and 255 inclusive. If not specified it defaults to the value in the getdns context.
- `version`: an integer betwen 0 and 255 inclusive. If not specified it defaults to 0.
- `do_bit`: must be either 0 or 1. If not specified it defaults to the value in the getdns context.
- `options`: a list containing dictionaries for each option to be specified. Each dictionary contains two keys: `option_code` (an integer) and `option_data` (in the form appropriate for that option code).

It is very important to note that the OPT resource record specified in the `add_opt_parameters extension` might not be the same the one that the API sends in the query. For example, if the application also includes any of the DNSSEC extensions, the API will make sure that the OPT resource record sets the resource record appropriately, making the needed changes to the settings from the `add_opt_parameters` extension.

The `client_subnet.py` program in our example directory shows how to pack and send an OPT record.

Getting Warnings for Responses that Violate the DNS Standard

To receive a warning if a particular response violates some parts of the DNS standard, use the `add_warning_for_bad_dns` extension. The extension's value is set to `getdns.EXTENSION_TRUE` to cause each reply in the `replies_tree` to contain an additional name, `bad_dns` (a list). The list is zero or more values that indicate types of bad DNS found in that reply. The list of values is:

`getdns.`**`BAD_DNS_CNAME_IN_TARGET`**

A DNS query type that does not allow a target to be a CNAME pointed to a CNAME

`getdns.`**`BAD_DNS_ALL_NUMERIC_LABEL`**

One or more labels in a returned domain name is all-numeric; this is not legal for a hostname

`getdns.`**`BAD_DNS_CNAME_RETURNED_FOR_OTHER_TYPE`**

A DNS query for a type other than CNAME returned a CNAME response

Using other class types

The vast majority of DNS requests are made with the Internet (IN) class. To make a request in a different DNS class, use, the `specify_class extension`. The extension's value (an int) contains the class number. Few applications will ever use this extension.

Extensions relating to the API

An application might want to see debugging information for queries, such as the length of time it takes for each query to return to the API. Use the `return_call_reporting` extension. The extension's value is set to `getdns.EXTENSION_TRUE` to add the name `call_reporting` (a list) to the top level of the `response` object. Each member of the list is a dict that represents one call made for the call to the API. Each member has the following names:

- `query_name` is the name that was sent
- `query_type` is the type that was queried for
- `query_to` is the address to which the query was sent
- `start_time` is the time the query started in milliseconds since the epoch, represented as an integer
- `end_time` is the time the query was received in milliseconds since the epoch, represented as an integer
- `entire_reply` is the entire response received
- `dnssec_result` is the DNSSEC status, or `getdns.DNSSEC_NOT_PERFORMED` if DNSSEC validation was not performed

Asynchronous queries

The getdns Python bindings support asynchronous queries, in which a query returns immediately and a callback function is invoked when the response data are returned. The query method interfaces are fundamentally the same, with a few differences:

- The query returns a transaction id. That transaction id may be used to cancel future callbacks
- The query invocation includes the name of a callback function. For example, if you'd like to call the function "my_callback" when the query returns, an address lookup could look like

```
>>> c = getdns.Context()
>>> tid = c.address('www.example.org', callback=my_callback)
```

- We've introduced a new `Context` method, called `run`. When your program is ready to check to see whether or not the query has returned, invoke the run() method on your context. Note that we use the libevent asynchronous event library and an event_base is associated with a context. So, if you have multiple outstanding events associated with a particular context, `run` will invoke all of those that are waiting and ready.
- In previous releases the callback argument took the form of a literal string, but as of this release you may pass in the name of any Python runnable, without quotes. The newer form is preferred.

The callback script takes four arguments: `type`, `result`, `userarg`, and `transaction_id. The ``type` argument contains the callback type, which may have one of the following values:

- `getdns.CALLBACK_COMPLETE`: The query was successful and the results are contained in the `result` argument
- `getdns.CALLBACK_CANCEL`: The callback was cancelled before the results were processed

- `getdns.CALLBACK_TIMEOUT`: The query timed out before the results were processed
- `getdns.CALLBACK_ERROR`: An unspecified error occurred

The `result` argument contains a result object, with the query response

The `userarg` argument contains the optional user argument that was passed to the query at the time it was invoked.

The `transaction_id` argument contains the transaction_id associated with a particular query; this is the same transaction id that was returned when the query was invoked.

This is an example callback function:

```
def cbk(type, result, userarg, tid):
    if type == getdns.CALLBACK_COMPLETE:
        status = result.status
        if status == getdns.RESPSTATUS_GOOD:
            for addr in result.just_address_answers:
                addr_type = addr['address_type']
                addr_data = addr['address_data']
                print '{0}: {1} {2}'.format(userarg, addr_type, addr_data)
        elif status == getdns.RESPSTATUS_NO_SECURE_ANSWERS:
            print "{0}: No DNSSEC secured responses found".format(hostname)
        else:
            print "{0}: getdns.address() returned error: {1}".format(hostname, status)
    elif type == getdns.CALLBACK_CANCEL:
        print 'Callback cancelled'
    elif type == getdns.CALLBACK_TIMEOUT:
        print 'Query timed out'
    else:
        print 'Unknown error'
```

5.2 `getdns` response data

5.2.1 Response data from queries

class `getdns.`**`Result`**

A getdns query (`Context.address()`, `Context.hostname()`, `Context.service()`, and `Context.general()`) returns a Result object. A Result object is only returned from a query and may not be instantiated by the programmer. It is a read-only object. Contents may not be overwritten or deleted.

It has no methods but includes the following attributes:

`status`

The `status` attribute contains the status code returned by the query. Note that it may be the case that the query can be successful but there are no data matching the query parameters. Programmers using this API will need to first check to see if the query itself was successful, then check for the records returned.

The `status` attribute may have the following values:

`getdns.`**`RESPSTATUS_GOOD`**

At least one response was returned

`getdns.`**`RESPSTATUS_NO_NAME`**

Queries for the name yielded all negative responses

getdns.**RESPSTATUS_ALL_TIMEOUT**
All queries for the name timed out

getdns.**RESPSTATUS_NO_SECURE_ANSWERS**
The context setting for getting only secure responses was specified, and at least one DNS response was received, but no DNS response was determined to be secure through DNSSEC.

RESPSTATUS_ALL_BOGUS_ANSWERS

The context setting for getting only secure responses was specified, and at least one DNS response was received, but all received responses for the requested name were bogus.

answer_type
The `answer_type` attribute contains the type of data that are returned (i.e., the namespace). The `answer_type` attribute may have the following values:

getdns.**NAMETYPE_DNS**
Normal DNS (**RFC 1035**)

getdns.**NAMETYPE_WINS**
The WINS name service (some reference needed)

canonical_name
The value of `canonical_name` is the name that the API used for its lookup. It is in FQDN presentation format.

just_address_answers
If the call was `address()`, the attribute `just_address_answers` (a list) is non-null. The value of `just_address_answers` is a list that contains all of the A and AAAA records from the `answer` sections of any of the replies, in the order they appear in the replies. Each item in the list is a dict with at least two names: `address_type` (a string whose value is either "IPv4" or "IPv6") and `address_data` (whose value is a string representation of an IP address). Note that the `dnssec_return_only_secure` extension affects what will appear in the just_address_answers list. Also note if later versions of the DNS return other address types, those types will appear in this list as well.

replies_full
The `replies_full` attribute is a Python dictionary containing the entire set of records returned by the query.

The following lists the status codes for response objects. Note that, if the status is that there are no responses for the query, the lists in `replies_full` and `replies_tree` will have zero length.

The top level of `replies_tree` can optionally have the following names: `canonical_name`, `intermediate_aliases` (a list), `answer_ipv4_address` `answer_ipv6_address`, and `answer_type` (an integer constant.).

- The value of `canonical_name` is the name that the API used for its lookup. It is in FQDN presentation format.
- The values in the `intermediate_aliases` list are domain names from any CNAME or unsynthesized DNAME found when resolving the original query. The list might have zero entries if there were no CNAMEs in the path. These may be useful, for example, for name comparisons when following the rules in RFC 6125.
- The value of `answer_ipv4_address` and `answer_ipv6_address` are the addresses of the server from which the answer was received.
- The value of `answer_type` is the type of name service that generated the response. The values are:

If the call was `address()`, the top level of `replies_tree` has an additional name, `just_address_answers` (a list). The value of `just_address_answers` is a list that contains all of the A and AAAA records from the `answer` sections of any of the replies, in the order they appear in the replies. Each item in the list is a dict with at least two names: `address_type` (a string whose value is either "IPv4" or "IPv6") and `address_data` (whose value is a string representation of an IP address). Note that the `dnssec_return_only_secure` extension affects what will appear in the just_address_answers list. Also note if later versions of the DNS return other address types, those types will appear in this list as well.

The API can make service discovery through SRV records easier. If the call was `service()`, the top level of `replies_tree has` an additional name, `srv_addresses` (a list). The list is ordered by priority and weight based on the weighting algorithm in **RFC 2782**, lowest priority value first. Each element of the list is a dictionary that has at least two names: `port` and `domain_name`. If the API was able to determine the address of the target domain name (such as from its cache or from the Additional section of responses), the dict for an element will also contain `address_type` (whose value is currently either "IPv4" or "IPv6") and `address_data` (whose value is a string representation of an IP address). Note that the `dnssec_return_only_secure` extension affects what will appear in the `srv_addresses` list.

`validation_chain`

The `validation_chain` attribute is a Python list containing the set of DNSSEC-related records needed for validation of a particular response. This set comes as validation_chain (a list) at the top level of the response object. This list includes all resource record dicts for all the resource records (DS, DNSKEY and their RRSIGs) that are needed to perform the validation from the root up.

`call_reporting`

A list of dictionaries containing call_debugging information, if requested in the query.

`replies_tree`

The names in each entry in the the `replies_tree` list for DNS responses include `header` (a dict), `question` (a dict), `answer` (a list), `authority` (a list), and `additional` (a list), corresponding to the sections in the DNS message format. The `answer`, `authority`, and `additional` lists each contain zero or more dicts, with each dict in each list representing a resource record.

The names in the `header` dict are all the fields from **RFC 1035#section-4.1.1**. They are: `id`, `qr`, `opcode`, `aa`, `tc`, `rd`, `ra`, `z`, `rcode`, `qdcount`, `ancount`, `nscount`, and `arcount`. All are integers.

The names in the `question` dict are the three fields from **RFC 1035#section-4.1.2**: `qname`, `qtype`, and `qclass`.

Resource records are a bit different than headers and question sections in that the RDATA portion often has its own structure. The other names in the resource record dictionaries are `name`, `type`, `class`, `ttl`, and `rdata` (which is a dict); there is no name equivalent to the RDLENGTH field. The OPT resource record does not have the `class` and the `ttl` name, but instead provides `udp_payload_size`, `extended_rcode`, `version`, `do`, and `z`.

The `rdata` dictionary has different names for each response type. There is a complete list of the types defined in the API. For names that end in "-obsolete" or "-unknown", the data are the entire RDATA field. For example, the `rdata` for an A record has a name `ipv4_address`; the rdata for an SRV record has the names `priority`, `weight`, `port`, and `target`.

Each rdata dict also has a `rdata_raw` element. This is useful for types not defined in this version of the API. It also might be of value if a later version of the API allows for additional parsers. Thus, doing a query for types not known by the API still will return a result: an `rdata` with just a `rdata_raw`.

It is expected that later extensions to the API will give some DNS types different names. It is also possible that later extensions will change the names for some of the DNS types listed above.

For example, a response to a Context.address() call for www.example.com would look something like this:

```
{     # This is the response object
 "replies_full": [ <bindata of the first response>, <bindata of the second response>␣
↪],
 "just_address_answers":
 [
   {
     "address_type": <bindata of "IPv4">,
     "address_data": <bindata of 0x0a0b0c01>,
   },
   {
     "address_type": <bindata of "IPv6">,
     "address_data": <bindata of 0x33445566334455663344556633445566>
   }
 ],
 "canonical_name": <bindata of "www.example.com">,
 "answer_type": NAMETYPE_DNS,
 "intermediate_aliases": [],
 "replies_tree":
 [
   {     # This is the first reply
     "header": { "id": 23456, "qr": 1, "opcode": 0, ... },
     "question": { "qname": <bindata of "www.example.com">, "qtype": 1, "qclass": 1 },
     "answer":
     [
       {
         "name": <bindata of "www.example.com">,
         "type": 1,
         "class": 1,
         "ttl": 33000,
         "rdata":
         {
           "ipv4_address": <bindata of 0x0a0b0c01>
           "rdata_raw": <bindata of 0x0a0b0c01>
         }
       }
     ],
     "authority":
     [
       {
         "name": <bindata of "ns1.example.com">,
         "type": 1,
         "class": 1,
         "ttl": 600,
         "rdata":
         {
           "ipv4_address": <bindata of 0x65439876>
           "rdata_raw": <bindata of 0x65439876>
         }
       }
     ]
     "additional": [],
     "canonical_name": <bindata of "www.example.com">,
     "answer_type": NAMETYPE_DNS
   },
   {     # This is the second reply
     "header": { "id": 47809, "qr": 1, "opcode": 0, ... },
     "question": { "qname": <bindata of "www.example.com">, "qtype": 28, "qclass": 1 }
 ↪,
     "answer":
     [
       {
          "name": <bindata of "www.example.com">,
          "type": 28,
          "class": 1,
          "ttl": 1000,
          "rdata":
          {
            "ipv6_address": <bindata of 0x33445566334455663344556633445566>
            "rdata_raw": <bindata of 0x33445566334455663344556633445566>
          }
       }
     ],
     "authority": [  # Same as for other record... ]
     "additional": [],
   },
 ]
}
```

5.2.2 Return Codes

The return codes for all the functions are:

getdns.**RETURN_GOOD**
Good

getdns.**RETURN_GENERIC_ERROR**
Generic error

getdns.**RETURN_BAD_DOMAIN_NAME**
Badly-formed domain name in first argument

getdns.**RETURN_BAD_CONTEXT**
The context has internal deficiencies

getdns.**RETURN_CONTEXT_UPDATE_FAIL**
Did not update the context

getdns.**RETURN_UNKNOWN_TRANSACTION**
An attempt was made to cancel a callback with a transaction_id that is not recognized

getdns.**RETURN_NO_SUCH_LIST_ITEM**
A helper function for lists had an index argument that was too high.

getdns.**RETURN_NO_SUCH_DICT_NAME**
A helper function for dicts had a name argument that for a name that is not in the dict.

getdns.**RETURN_WRONG_TYPE_REQUESTED**
A helper function was supposed to return a certain type for an item, but the wrong type was given.

getdns.**RETURN_NO_SUCH_EXTENSION**
A name in the extensions dict is not a valid extension.

getdns.**RETURN_EXTENSION_MISFORMAT**
One or more of the extensions have a bad format.

getdns.**RETURN_DNSSEC_WITH_STUB_DISALLOWED**
A query was made with a context that is using stub resolution and a DNSSEC extension specified.

getdns.**RETURN_MEMORY_ERROR**
Unable to allocate the memory required.

getdns.**RETURN_INVALID_PARAMETER**
A required parameter had an invalid value.

getdns.**RETURN_NOT_IMPLEMENTED**
The requested API feature is not implemented.

5.3 getdns exceptions

5.3.1 getdns exceptions

exception getdns.**error**
getdns will throw an exception, getdns.error, under certain conditions. Those conditions include:

- a required parameter having a bad value
- a badly-formed domain name in the query
- a bad Context() object
- a failed Context() update
- an out-of-bounds error for a getdns data structure
- requesting an extension that doesn't exist
- requesting DNSSEC validation while using stub resolution

Please note that a successful return from a getdns method does *not* indicate that the query returned the records being requested, but rather that the query is formed correctly and has been submitted to the DNS. A getdns exception is typically the result of a coding error.

getdns will set the exception message to a diagnostic string, which may be examined for help in resolving the error.

5.3.2 Example

```
import getdns, sys

c = getdns.Context()
try:
    results = c.address('www.example.com', foo='bar')
except getdns.error, e:
    print(str(e))
    sys.exit(1)
```

This will result in "A required parameter had an invalid value" being printed to the screen.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

getdns, 18

Index

Symbols

__version__, 9
__version__ (in module getdns), 14

A

address() (getdns.Context method), 14
alabel_to_ulabel(), 9
answer_type (getdns.Result attribute), 19
append_name (getdns.Context attribute), 11

B

BAD_DNS_ALL_NUMERIC_LABEL (in module getdns), 16
BAD_DNS_CNAME_IN_TARGET (in module getdns), 16
BAD_DNS_CNAME_RETURNED_FOR_OTHER_TYPE (in module getdns), 16

C

call_reporting (getdns.Result attribute), 20
canonical_name (getdns.Result attribute), 19
Context (class in getdns), 11

D

dns_root_servers (getdns.Context attribute), 11
dns_transport_list (getdns.Context attribute), 12
dnssec_allowed_skew (getdns.Context attribute), 12
dnssec_trust_anchors (getdns.Context attribute), 12

E

edns_client_subnet_private (getdns.Context attribute), 12
edns_do_bit (getdns.Context attribute), 12
edns_extended_rcode (getdns.Context attribute), 12
edns_maximum_udp_payload_size (getdns.Context attribute), 12
edns_version (getdns.Context attribute), 12

F

follow_redirects (getdns.Context attribute), 12

G

general() (getdns.Context method), 13
get_api_information() (getdns.Context method), 14
get_errorstr_by_id(), 9
get_supported_attributes() (getdns.Context method), 14
getdns (module), 11, 18, 23
getdns.error, 23
getdns.RESPSTATUS_ALL_BOGUS_ANSWERS (in module getdns), 19

H

hostname() (getdns.Context method), 14

I

idle_timeout (getdns.Context attribute), 12
implementation_string (getdns.Context attribute), 12

J

just_address_answers (getdns.Result attribute), 19

L

limit_outstanding_queries (getdns.Context attribute), 12

N

namespaces (getdns.Context attribute), 12

R

replies_full (getdns.Result attribute), 19
replies_tree (getdns.Result attribute), 20
resolution_type (getdns.Context attribute), 13
Result (class in getdns), 18
Result.getdns.NAMETYPE_DNS (in module getdns), 19
Result.getdns.NAMETYPE_WINS (in module getdns), 19
Result.getdns.RESPSTATUS_ALL_TIMEOUT (in module getdns), 18
Result.getdns.RESPSTATUS_GOOD (in module getdns), 18

Result.getdns.RESPSTATUS_NO_NAME (in module getdns), 18
Result.getdns.RESPSTATUS_NO_SECURE_ANSWERS (in module getdns), 19
RETURN_BAD_CONTEXT (in module getdns), 22
RETURN_BAD_DOMAIN_NAME (in module getdns), 22
RETURN_CONTEXT_UPDATE_FAIL (in module getdns), 22
RETURN_DNSSEC_WITH_STUB_DISALLOWED (in module getdns), 22
RETURN_EXTENSION_MISFORMAT (in module getdns), 22
RETURN_GENERIC_ERROR (in module getdns), 22
RETURN_GOOD (in module getdns), 22
RETURN_INVALID_PARAMETER (in module getdns), 22
RETURN_MEMORY_ERROR (in module getdns), 22
RETURN_NO_SUCH_DICT_NAME (in module getdns), 22
RETURN_NO_SUCH_EXTENSION (in module getdns), 22
RETURN_NO_SUCH_LIST_ITEM (in module getdns), 22
RETURN_NOT_IMPLEMENTED (in module getdns), 23
RETURN_UNKNOWN_TRANSACTION (in module getdns), 22
RETURN_WRONG_TYPE_REQUESTED (in module getdns), 22
RFC
    RFC 1035, 19
    RFC 1035#section-4.1.1, 20
    RFC 1035#section-4.1.2, 20
    RFC 2671, 16
    RFC 2782, 20
    RFC 4343#section-2.1, 13
root_trust_anchor(), 9

S

service() (getdns.Context method), 14
status (getdns.Result attribute), 18
suffix (getdns.Context attribute), 13

T

timeout (getdns.Context attribute), 13
tls_authentication (getdns.Context attribute), 13
tls_query_padding_blocksize (getdns.Context attribute), 13

U

ulabel_to_alabel(), 9
upstream_recursive_servers (getdns.Context attribute), 13

V

validation_chain (getdns.Result attribute), 20
version_string (getdns.Context attribute), 13