zenoh-python

Release 0.7.2-rc

Jun 07, 2023

Contents

[Zenoh](https://zenoh.io) /zeno/ is a stack that unifies data in motion, data at rest and computations. It elegantly blends traditional pub/sub with geo distributed storage, queries and computations, while retaining a level of time and space efficiency that is well beyond any of the mainstream stacks.

Before delving into the examples, we need to introduce few **Zenoh** concepts. First off, in Zenoh you will deal with **Resources**, where a resource is made up of a key and a value. The other concept you'll have to familiarize yourself with are **key expressions**, such as `robot/sensor/temp`, `robot/sensor/*`, `robot/**`, etc. As you can gather, the above key expression denotes set of keys, while the `*` and `**` are wildcards representing respectively (1) a single chunk (non-empty sequence of characters that doesn't contain `/`), and (2) any amount of chunks (including 0).

Below are some examples that highlight these key concepts and show how easy it is to get started with.

CHAPTER 1

Quick start examples:

1.1 Publish a key/value pair onto Zenoh

```
>>> import zenoh
>>> z = zenoh.open()
>>> z.put('/demo/example/hello', 'Hello World!')
```

1.2 Subscribe to a set of keys with Zenoh

```
>>> import zenoh, time
>>> def listener(sample):
>>>     print(f"{sample.key_expr} => {sample.payload.decode('utf-8')}")
>>>
>>> z = zenoh.open()
>>> subscriber = z.subscribe('/demo/example/**', listener)
>>> time.sleep(60)
>>> subscriber.undeclare()
```

1.3 Get keys/values from zenoh

```
>>> import zenoh
>>> z = zenoh.open()
>>> for response in z.get('/demo/example/**', zenoh.Queue()):
>>>     response = response.ok
>>>     print(f"{response.key_expr} => {response.payload.decode('utf-8')}")
```

1.3.1 module zenoh

`zenoh.`**`init_logger`**`()`

Initialize the logger used by the Rust implementation of this API.

Once initialized, you can configure the logs displayed by the API using the `RUST_LOG` environment variable. For instance, start python with the *debug* logs available:

```
$ RUST_LOG=debug python
```

More details on the RUST_LOG configuration on https://docs.rs/env_logger/latest/env_logger

`zenoh.`**`open`**`(`**args*, ***kwargs*`)`

Open a zenoh-net Session.

Parameters **`config`** (`Config`) – The configuration of the zenoh-net session

Return type *Session*

Example

```
>>> import zenoh
>>> s = zenoh.open(zenoh.Config())
```

`zenoh.`**`scout`**`(`*handler: Union[zenoh.closures.IHandler[zenoh.value.Hello, typing.Any, typing.Any][zenoh.value.Hello, Any, Any], zenoh.closures.IClosure[zenoh.value.Hello, typing.Any][zenoh.value.Hello, Any], Tuple[zenoh.closures.IClosure, Any], Tuple[Callable[[zenoh.value.Hello], Any], Callable[[], None], Any], Tuple[Callable[[zenoh.value.Hello], Any], Callable[[], None]], Callable[[zenoh.value.Hello], Any]] = None, what: str = None, config: zenoh.config.Config = None, timeout=None*`)`

Scout for routers and/or peers.

This spawns a task that periodically sends scout messages for a specified duration and returns a list of received `Hello` messages.

Parameters

- **`what`** – The kind of zenoh process to scout for
- **`config`** – The configuration to use for scouting
- **`timeout`** – the duration of scout (in seconds)
- **`handler`** –

Return type list of `Hello`

Example

```
>>> import zenoh
>>> for hello in zenoh.scout(what = "peer|router", timeout=1.0).receiver():
...     print(hello)
```

Hello

`class` `zenoh.`**`Hello`**

Represents a single Zenoh node discovered through scouting.

`locators`

The locators through which this node may be adressed.

whatami
The node's type, returning either None, 'peer', 'router', or 'client'.

zid
The node's Zenoh UUID.

Config

class zenoh.**Config**

static from_file (*filename: str*)
Reads the configuration from a file. The file's extension must be json, json5 or yaml.

static from_json5 (*json: str*)
Reads the configuration from a JSON5 string.

JSON5 is a superset of JSON, so any JSON string is a valid input for this function.

static from_obj (*obj*)
Reads the configuration from `obj` as if it was a JSON file.

get_json (*path: str*) → str
Returns the part of the configuration at `path`, in a JSON-serialized form.

insert_json5 (*path: str*, *value: str*) → str
Inserts the provided value (read from JSON) at the given path in the configuration.

Session

class zenoh.**Session**
A Zenoh Session, the core interraction point with a Zenoh network.

Note that most applications will only need a single instance of `Session`. You should _never_ construct one session per publisher/subscriber, as this will significantly increase the size of your Zenoh network, while preventing potential locality-based optimizations.

close ()
Attempts to close the Session.

The session will only be closed if all publishers, subscribers and queryables based on it have been undeclared, and there are no more python references to it.

config () → zenoh.config.Config
Returns a configuration object that can be used to alter the session's configuration at runtime.

Note that in Python specifically, the config you passed to the session becomes the result of this function if you passed one, letting you keep using it.

declare_keyexpr (*keyexpr: Union[KeyExpr, _KeyExpr, str]*) → zenoh.keyexpr.KeyExpr
Informs Zenoh that you intend to use the provided Key Expression repeatedly.

This function returns an optimized representation of the passed `keyexpr`.

It is generally not needed to declare key expressions, as declaring a subscriber, a queryable, or a publisher will also inform Zenoh of your intent to use their key expressions repeatedly.

declare_publisher (*keyexpr: Union[KeyExpr, _KeyExpr, str], priority: zenoh.enums.Priority = None, congestion_control: zenoh.enums.CongestionControl = None*)
Declares a publisher, which may be used to send values repeatedly onto a same key expression.

Written resources that match the given key will only be sent on the network if matching subscribers exist in the system.

Parameters

- **`keyexpr`** – The key expression to publish to
- **`priority`** – The priority to use when routing the published data
- **`congestion_control`** – The congestion control to use when routing the published data

Return type *Publisher*

Examples

```
>>> import zenoh
>>> s = zenoh.open({})
>>> pub = s.declare_publisher('key/expression')
>>> pub.put('value')
```

`declare_pull_subscriber`(*keyexpr: Union[KeyExpr, _KeyExpr, str], handler: Union[zenoh.closures.IHandler[zenoh.value.Sample, typing.Any, typing.Any][zenoh.value.Sample, Any, Any], zenoh.closures.IClosure[zenoh.value.Sample, typing.Any][zenoh.value.Sample, Any], Tuple[zenoh.closures.IClosure, Any], Tuple[Callable[[zenoh.value.Sample], Any], Callable[[], None], Any], Tuple[Callable[[zenoh.value.Sample], Any], Callable[[], None]], Callable[[zenoh.value.Sample], Any]], reliability: zenoh.enums.Reliability = None*) → zenoh.session.PullSubscriber

Declares a pull-mode subscriber, which will receive a single published sample with a key expression intersecting `keyexpr` any time its `pull` method is called.

These samples are passed to the *handler*'s closure as instances of the *Sample* class. The *handler* can typically be a queue or a callback. The *handler*'s receiver is returned as the *receiver* field of the returned *PullSubscriber*.

Parameters

- **`keyexpr`** – The key expression to subscribe to
- **`handler`** –
- **`reliability`** – the reliability to use when routing the subscribed samples

Return type *PullSubscriber*

Examples

```
>>> import zenoh
>>> s = zenoh.open({})
>>> sub = s.declare_pull_subscriber('key/expression', lambda sample:
...     print(f"Received '{sample.key_expr}': '{sample.payload.decode('utf-8
↪')}'"))
...
>>> sub.pull()
```

declare_queryable(*keyexpr: Union[KeyExpr, _KeyExpr, str], handler: Union[zenoh.closures.IHandler[zenoh.queryable.Query, typing.Any, typing.Any][zenoh.queryable.Query, Any, Any], zenoh.closures.IClosure[zenoh.queryable.Query, typing.Any][zenoh.queryable.Query, Any], Tuple[zenoh.closures.IClosure, Any], Tuple[Callable[[zenoh.queryable.Query], Any], Callable[[], None], Any], Tuple[Callable[[zenoh.queryable.Query], Any], Callable[[], None]], Callable[[zenoh.queryable.Query], Any]], complete: bool = None*)

Declares a queryable, which will receive queries intersecting with `keyexpr`.

These queries are passed to the *handler* as instances of the *Query* class. The *handler* can typically be a queue or a callback. The *handler*'s receiver is returned as the *receiver* field of the returned *Queryable*. The replies can be sent back by calling the *reply'function of the 'Query*.

Examples

Using a callback:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> qabl = s.declare_queryable('key/expression', lambda query:
...     query.reply(zenoh.Sample('key/expression', 'value')))
```

Using a queue:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> qabl = s.declare_queryable('key/expression', zenoh.Queue())
>>> while True:
...     query = qabl.receiver.get()
...     query.reply(zenoh.Sample('key/expression', 'value'))
...     del query
```

IMPORTANT: due to how RAII and Python work, you MUST bind this function's return value to a variable in order for it to function as expected. This is because as soon as a value is no longer referenced in Python, that value's destructor will run, which will undeclare your queryable, stopping it immediately.

declare_subscriber(*keyexpr: Union[KeyExpr, _KeyExpr, str], handler: Union[zenoh.closures.IHandler[zenoh.value.Sample, typing.Any, typing.Any][zenoh.value.Sample, Any, Any], zenoh.closures.IClosure[zenoh.value.Sample, typing.Any][zenoh.value.Sample, Any], Tuple[zenoh.closures.IClosure, Any], Tuple[Callable[[zenoh.value.Sample], Any], Callable[[], None], Any], Tuple[Callable[[zenoh.value.Sample], Any], Callable[[], None]], Callable[[zenoh.value.Sample], Any]], reliability: zenoh.enums.Reliability = None*) → zenoh.session.Subscriber

Declares a subscriber, which will receive any published sample with a key expression intersecting `keyexpr`.

These samples are provided to the *handler* as instances of the *Sample* class. The *handler* can typically be a queue or a callback. The *handler*'s receiver is returned as the *receiver* field of the returned *Subscriber*.

Parameters

- **keyexpr** – The key expression to subscribe to
- **handler** –
- **reliability** – the reliability to use when routing the subscribed samples

Return type *Subscriber*

Examples

Using a callback:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> sub = s.declare_subscriber('key/expression', lambda sample:
...     print(f"Received '{sample.key_expr}': '{sample.payload.decode('utf-8
↪')}'"))
```

Using a queue:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> sub = s.declare_subscriber('key/expression', zenoh.Queue())
>>> for sample in sub.receiver:
>>>     print(f"{sample.key_expr}: {sample.payload.decode('utf-8')}")
```

IMPORTANT: due to how RAII and Python work, you MUST bind this function's return value to a variable in order for it to function as expected. This is because as soon as a value is no longer referenced in Python, that value's destructor will run, which will undeclare your subscriber, deactivating the subscription immediately.

delete (*keyexpr: Union[KeyExpr, _KeyExpr, str], priority: zenoh.enums.Priority = None, congestion_control: zenoh.enums.CongestionControl = None*)

Deletes the values associated with the keys included in `keyexpr`.

This uses the same mechanisms as `session.put`, and will be received by subscribers. This operation is especially useful with storages.

Parameters

- **keyexpr** – The key expression to publish
- **priority** – The priority to use when routing the delete
- **congestion_control** – The congestion control to use when routing the delete

Examples

```
>>> import zenoh
>>> s = zenoh.open({})
>>> s.delete('key/expression')
```

get (*selector: Union[Selector, _Selector, KeyExpr, _KeyExpr, str], handler: Union[zenoh.closures.IHandler[zenoh.value.Reply, typing.Any, ~Receiver][zenoh.value.Reply, Any, Receiver], zenoh.closures.IClosure[zenoh.value.Reply, typing.Any][zenoh.value.Reply, Any], Tuple[zenoh.closures.IClosure, Receiver], Tuple[Callable[[zenoh.value.Reply], Any], Callable[[], None], Receiver], Tuple[Callable[[zenoh.value.Reply], Any], Callable[[], None]], Callable[[zenoh.value.Reply], Any]], consolidation: zenoh.enums.QueryConsolidation = None, target: zenoh.enums.QueryTarget = None, value: Union[zenoh.value.IValue, bytes, str, int, float, object] = None*) → Receiver

Emits a query, which queryables with intersecting selectors will be able to reply to.

The replies are provided to the given *handler* as instances of the *Reply* class. The *handler* can typically be a queue, a single callback or a pair of callbacks. The *handler*'s receiver is returned by the *get* function.

Parameters

- **selector** – The selection of keys to query
- **handler** –

- **consolidation** – The consolidation to apply to replies
- **target** – The queryables that should be target to this query
- **value** – An optional value to attach to this query

Returns The receiver of the handler

Return type Receiver

Examples

Using a queue:

```
>>> import zenoh
>>> s = zenoh.open({})
>>> for reply in s.get('key/expression', zenoh.Queue()):
...     try:
...         print(f"Received '{reply.ok.key_expr}': '{reply.ok.payload.decode(
↪'utf-8')}'")
...     except:
...         print(f"Received ERROR: '{reply.err.payload.decode('utf-8')}'")
```

Using a single callback:

```
>>> s.get('key/expression', lambda reply:
...     print(f"Received '{reply.ok.key_expr}': '{reply.ok.payload.decode(
↪'utf-8')}'")
...     if reply.ok is not None else print(f"Received ERROR: '{reply.err.
↪payload.decode('utf-8')}'"))
```

Using a reply callback and a termination callback:

```
>>> s.get('key/expression', (
...     lambda reply:
...         print(f"Received '{reply.ok.key_expr}': '{reply.ok.payload.decode(
↪'utf-8')}'")
...         if reply.ok is not None else print(f"Received ERROR: '{reply.err.
↪payload.decode('utf-8')}'"),
...     lambda:
...         print("No more replies")))
```

info()

Returns an accessor for informations about this Session

put(*keyexpr: Union[KeyExpr, _KeyExpr, str], value: Union[zenoh.value.IValue, bytes, str, int, float, object], encoding=None, priority: zenoh.enums.Priority = None, congestion_control: zenoh.enums.CongestionControl = None, sample_kind: zenoh.enums.SampleKind = None*)

Sends a value over Zenoh.

Subscribers on an expression that intersect with `keyexpr` will receive the sample. Storages will store the value if `keyexpr` is non-wild, or update the values for all known keys that are included in `keyexpr` if it is wild.

Parameters

- **keyexpr** – The key expression to publish
- **value** – The value to send
- **priority** – The priority to use when routing the published data

- **congestion_control** – The congestion control to use when routing the published data
- **sample_kind** – The kind of sample to send

Examples

```
>>> import zenoh
>>> s = zenoh.open({})
>>> s.put('key/expression', 'value')
```

Info

class zenoh.**Info**(*session: _Session*)

peers_zid() → List[zenoh.value.ZenohId]
Returns the neighbooring peers' identifiers

routers_zid() → List[zenoh.value.ZenohId]
Returns the neighbooring routers' identifiers

zid() → zenoh.value.ZenohId
Returns this Zenoh Session's identifier

KeyExpr

class zenoh.**KeyExpr**

Zenoh's address space is designed around keys which serve as the names of ressources.

Keys are slash-separated lists of non-empty UTF8 strings. They may not contain the following characters: `$*#?`.

Zenoh's operations are executed on key expressions, a small language that allows the definition of sets of keys via the use of wildcards:

- `*` is the single-chunk wildcard, and will match any chunk: `a/*/c` will match `a/b/c`, `a/hello/c`, etc…
- `**` is the 0 or more chunks wildcard: `a/**/c` matches `a/c`, `a/b/c`, `a/b/hello/c`, etc…
- `$*` is the subchunk wildcard, it will match any amount of non-/ characters: `a/b$*` matches `a/b`, `a/because`, `a/blue`… but not `a/c` nor `a/blue/c`

To allow for better performance and gain the property that two key expressions define the same set if and only if they are the same string, the rules of canon form are mandatory for a key expression to be propagated by a Zenoh network:

- `**/**` may not exist, as it could always be replaced by the shorter `**`,
- `**/*` may not exist, and must be written as its equivalent `*/**` instead,
- `$*` may not exist alone in a chunk, as it must be written `*` instead.

The `KeyExpr.autocanonize` constructor exists to correct eventual infrigements of the canonization rules.

A KeyExpr is a string that has been validated to be a valid Key Expression.

static autocanonize(*expr: str*) → zenoh.keyexpr.KeyExpr
This alternative constructor for key expressions will attempt to canonize the passed expression before checking if it is valid.

Raises a zenoh.ZError exception if `expr` is not a valid key expression.

includes(*other: Union[KeyExpr, _KeyExpr, str]*) → bool
This method returns `True` if all of the keys defined by `other` also belong to the set defined by `self`.

intersects(*other: Union[KeyExpr, _KeyExpr, str]*) → bool
This method returns `True` if there exists at least one key that belongs to both sets defined by `self` and `other`.

undeclare(*session: Session*)
Undeclares a key expression previously declared on the session.

Sample

class zenoh.**Sample**
A KeyExpr-Value pair, annotated with the kind (PUT or DELETE) of publication used to emit it and a timestamp.

encoding
A shortcut to `self.value.encoding`

key_expr
The sample's key expression

kind
The sample's kind

payload
A shortcut to `self.value.payload`

timestamp
The sample's timestamp. May be None.

value
The sample's value

SampleKind

class zenoh.**SampleKind**
Similar to an HTTP METHOD: only PUT and DELETE are currently supported.

static DELETE() → zenoh.enums.SampleKind

static PUT() → zenoh.enums.SampleKind

Value

class zenoh.**Value**
A Value is a pair of a binary payload, and a mime-type-like encoding string.

When constructed with `encoding==None`, the encoding will be selected depending on the payload's type.

static autoencode(*value: Union[zenoh.value.IValue, bytes, str, int, float, object]*) → zenoh.value.Value
Automatically encodes the value based on its type

Encoding

class zenoh.**Encoding**

> **static APP_CUSTOM**() → zenoh.enums.Encoding
>
> **static APP_FLOAT**() → zenoh.enums.Encoding
>
> **static APP_INTEGER**() → zenoh.enums.Encoding
>
> **static APP_JSON**() → zenoh.enums.Encoding
>
> **static APP_OCTET_STREAM**() → zenoh.enums.Encoding
>
> **static APP_PROPERTIES**() → zenoh.enums.Encoding
>
> **static APP_SQL**() → zenoh.enums.Encoding
>
> **static APP_XHTML_XML**() → zenoh.enums.Encoding
>
> **static APP_XML**() → zenoh.enums.Encoding
>
> **static APP_X_WWW_FORM_URLENCODED**() → zenoh.enums.Encoding
>
> **static EMPTY**() → zenoh.enums.Encoding
>
> **static IMAGE_GIF**() → zenoh.enums.Encoding
>
> **static IMAGE_JPEG**() → zenoh.enums.Encoding
>
> **static IMAGE_PNG**() → zenoh.enums.Encoding
>
> **static TEXT_CSS**() → zenoh.enums.Encoding
>
> **static TEXT_CSV**() → zenoh.enums.Encoding
>
> **static TEXT_HTML**() → zenoh.enums.Encoding
>
> **static TEXT_JAVASCRIPT**() → zenoh.enums.Encoding
>
> **static TEXT_JSON**() → zenoh.enums.Encoding
>
> **static TEXT_PLAIN**() → zenoh.enums.Encoding
>
> **static TEXT_XML**() → zenoh.enums.Encoding
>
> **append**(*s: str*)
>
> **static from_str**(*s: str*) → zenoh.enums.Encoding

Publisher

class zenoh.**Publisher**(*p: _Publisher*)

> Use `Publisher` (constructed with `Session.declare_publisher`) when you want to send values often for the same key expression, as declaring them informs Zenoh that this is you intent, and optimizations will be set up to do so.
>
> **delete**()
>
> > An optimised version of `session.delete(self.key_expr)`
>
> **key_expr**
>
> > This `Publisher`'s key expression

put (*value: Union[zenoh.value.IValue, bytes, str, int, float, object], encoding: zenoh.enums.Encoding = None*)
An optimised version of `session.put(self.key_expr, value, encoding=encoding)`

undeclare ()
Stops the publisher.

CongestionControl

`class zenoh.`**CongestionControl**
Defines the network's behaviour regarding a message when heavily congested.

`static` **BLOCK** () → zenoh.enums.CongestionControl
Prevents the message from being dropped at all cost. In the face of heavy congestion on a part of the network, this could result in your publisher node blocking.

`static` **DROP** () → zenoh.enums.CongestionControl
Allows the message to be dropped if all buffers are full.

Priority

`class zenoh.`**Priority**
The priority of a sending operation.

They are ordered à la Linux priority: `Priority.REAL_TIME() < Priority.INTERACTIVE_HIGH() < Priority.INTERACTIVE_LOW() < Priority.DATA() < Priority.BACKGROUND()`

`static` **BACKGROUND** () → zenoh.enums.Priority

`static` **DATA** () → zenoh.enums.Priority

`static` **DATA_HIGH** () → zenoh.enums.Priority

`static` **DATA_LOW** () → zenoh.enums.Priority

`static` **INTERACTIVE_HIGH** () → zenoh.enums.Priority

`static` **INTERACTIVE_LOW** () → zenoh.enums.Priority

`static` **REAL_TIME** () → zenoh.enums.Priority

Subscriber

`class zenoh.`**Subscriber** (*s: _Subscriber*, *receiver=None*)
A handle to a subscription.

Its main purpose is to keep the subscription active as long as it exists.

When constructed through `Session.declare_subscriber(session, keyexpr, handler)`, it exposes `handler`'s receiver through `self.receiver`.

undeclare ()
Undeclares the subscription

PullSubscriber

class zenoh.**PullSubscriber**(*s: _PullSubscriber*, *receiver=None*)

A handle to a pull subscription.

Its main purpose is to keep the subscription active as long as it exists.

When constructed through `Session.declare_pull_subscriber(session, keyexpr, handler)`, it exposes `handler`'s receiver through `self.receiver`.

Calling `self.pull()` will prompt the Zenoh network to send a new sample when available.

pull()

Prompts the Zenoh network to send a new sample if available. Note that this sample will not be returned by this function, but provided to the handler's callback.

undeclare()

Undeclares the subscription

Reliability

class zenoh.**Reliability**

Used by subscribers to inform the network of the reliability it wishes to obtain.

static BEST_EFFORT() → zenoh.enums.CongestionControl

Informs the network that dropping some messages is acceptable

static RELIABLE() → zenoh.enums.CongestionControl

Informs the network that this subscriber wishes for all publications to reliably reach it.

Note that if a publisher puts a sample with the `CongestionControl.DROP()` option, this reliability requirement may be infringed to prevent slow readers from blocking the network.

Query

class zenoh.**Query**

decode_parameters() → Dict[str, str]

Decodes the value selector into a dictionary.

Raises a ZError if duplicate keys are found, as they might otherwise be used for HTTP Parameter Pollution like attacks.

key_expr

The query's targeted key expression

parameters

The query's value selector. If you'd rather not bother with parsing it yourself, use `self.decode_parameters()` instead.

reply(*sample: zenoh.value.Sample*)

Allows you to reply to a query. You may send any amount of replies to a single query, including 0.

selector

The query's selector as a whole.

value

The query's value.

Selector

class `zenoh.`**`Selector`**

A selector is the combination of a [Key Expression](crate::prelude::KeyExpr), which defines the set of keys that are relevant to an operation, and a `parameters`, a set of key-value pairs with a few uses:

- specifying arguments to a queryable, allowing the passing of Remote Procedure Call parameters
- filtering by value,
- filtering by metadata, such as the timestamp of a value,

When in string form, selectors look a lot like a URI, with similar semantics:

- the `key_expr` before the first `?` must be a valid key expression.
- the `parameters` after the first `?` should be encoded like the query section of a URL:
  - key-value pairs are separated by `&`,
  - the key and value are separated by the first =,
  - in the absence of =, the value is considered to be the empty string,
  - both key and value should use percent-encoding to escape characters,
  - defining a value for the same key twice is considered undefined behavior.

Zenoh intends to standardize the usage of a set of keys. To avoid conflicting with RPC parameters, the Zenoh team has settled on reserving the set of keys that start with non-alphanumeric characters.

This document will summarize the standardized keys for which Zenoh provides helpers to facilitate coherent behavior for some operations.

Queryable implementers are encouraged to prefer these standardized keys when implementing their associated features, and to prefix their own keys to avoid having conflicting keys with other queryables.

Here are the currently standardized keys for Zenoh:

- `_time`: used to express interest in only values dated within a certain time range, values for this key must be readable by the [Zenoh Time DSL](zenoh_util::time_range::TimeRange) for the value to be considered valid.
- `_filter`: *TBD* Zenoh intends to provide helper tools to allow the value associated with this key to be treated as a predicate that the value should fulfill before being returned. A DSL will be designed by the Zenoh team to express these predicates.

`decode_parameters``()` → Dict[str, str]

Decodes the value selector part of the selector.

Raises a ZError if some keys were duplicated: duplicated keys are considered undefined behaviour, but we encourage you to refuse to process incoming messages with duplicated keys, as they might be attempting to use HTTP Parameter Pollution like exploits.

`key_expr`

The key expression part of the selector.

`parameters`

The value selector part of the selector.

`set_parameters`

The value selector part of the selector.

QueryTarget

class zenoh.**QueryTarget**

QueryConsolidation

class zenoh.**QueryConsolidation**

Reply

class zenoh.**Reply**

A reply to a query (`Session.get`).

A single query can result in multiple replies from multiple queryables.

err

The reply's error value.

Raises a `ZError` if the `self` is actually an `ok` reply.

ok

The reply's inner data sample.

Raises a `ZError` if the `self` is actually an `err` reply.

replier_id

The reply's sender's id.

Queryable

class zenoh.**Queryable**(*inner: _Queryable*, *receiver*)

A handle to a queryable.

Its main purpose is to keep the queryable active as long as it exists.

When constructed through `Session.declare_queryable(session, keyexpr, handler)`, it exposes `handler`'s receiver through `self.receiver`.

undeclare()

Stops the queryable.

ZenohId

class zenoh.**ZenohId**

A Zenoh UUID

Timestamp

class zenoh.**Timestamp**

A timestamp taken from the Zenoh HLC (Hybrid Logical Clock).

These timestamps are guaranteed to be unique, as each machine annotates its perceived time with a UUID, which is used as the least significant part of the comparison operation.

`seconds_since_unix_epoch`
Returns the number of seconds since the Unix Epoch.

You shouldn't use this for comparison though, and rely on comparison operators between members of this class.

class `zenoh.`**`Queue`**(*bound: int = None*)
A binding for a Rust multi-producer, single-consumer queue implementation.

When used as a handler, it provides itself as the receiver, and will provide a callback that appends elements to the queue.

Can be bounded by passing a maximum size as `bound`.

`closure`
The part of the handler that should be passed as a callback to a zenoh function.

`get`(*timeout: float = None*)
Gets one element from the queue.

Raises a `StopIteration` exception if the queue was closed before the timeout ran out, this allows using the Queue as an iterator in for-loops. Raises a `TimeoutError` if the timeout ran out.

`get_remaining`(*timeout: float = None*) → List[In]
Awaits the closing of the queue, returning the remaining queued values in a list. The values inserted into the queue up until this happens will be available through `get`.

Raises a `TimeoutError` if the timeout in seconds provided was exceeded before closing, whose `args[0]` will contain the elements that were collected before timing out.

`put`(*value*)
Puts one element on the queue.

Raises a `PyBrokenPipeError` if the Queue has been closed.

`receiver`
The part of the handler that should be used as the receiver when the handler is channel-like.

class `zenoh.`**`ListCollector`**(*timeout=None*)
A simple collector that aggregates values into a list.

When used as a handler, it provides a callback that appends elements to a list, and provides a function that will await the closing of the callback before returning said list.

`closure`
The part of the handler that should be passed as a callback to a zenoh function.

`receiver`
The part of the handler that should be used as the receiver when the handler is channel-like.

class `zenoh.`**`Closure`**(*closure: Union[zenoh.closures.IHandler[~In, ~Out, typing.Any][In, Out, Any], zenoh.closures.IClosure[~In, ~Out][In, Out], Tuple[Callable[[In], Out], Callable[[], None]], Callable[[In], Out]], type_adaptor: Callable[[Any], In] = None, prevent_direct_calls=False*)
A Closure is a pair of a `call` function that will be used as a callback, and a `drop` function that will be called when the closure is destroyed.

`call`
Returns the closure's call function as a lambda.

`drop`
Returns the closure's destructor as a lambda.

class `zenoh.`**`Handler`**(*input: Union[zenoh.closures.IHandler[~In, ~Out, ~Receiver][In, Out, Receiver], zenoh.closures.IClosure[~In, ~Out][In, Out], Tuple[zenoh.closures.IClosure, Receiver], Tuple[Callable[[In], Out], Callable[[], None], Receiver], Tuple[Callable[[In], Out], Callable[[], None]], Callable[[In], Out]], type_adaptor: Callable[[Any], In] = None, prevent_direct_calls=True*)

A Handler is a value that may be converted into a callback closure for zenoh to use on one side, while possibly providing a receiver for the data that zenoh would provide through that callback.

Note that the values will be piped onto a `Queue` before being sent to your handler by another Thread unless either:

a) `input` is already an instance of `Closure` or `Handler` where `input.closure` is an instance of `Closure`

b) `prevent_direct_calls` is set to `False`

`closure`
The part of the handler that should be passed as a callback to a zenoh function.

`receiver`
The part of the handler that should be used as the receiver when the handler is channel-like.

class `zenoh.`**`IClosure`**

A Closure is a pair of a `call` function that will be used as a callback, and a `drop` function that will be called when the closure is destroyed.

`call`
Returns the closure's call function as a lambda.

`drop`
Returns the closure's destructor as a lambda.

class `zenoh.`**`IHandler`**

A Handler is a value that may be converted into a callback closure for zenoh to use on one side, while possibly providing a receiver for the data that zenoh would provide through that callback.

`closure`
The part of the handler that should be passed as a callback to a zenoh function.

`receiver`
The part of the handler that should be used as the receiver when the handler is channel-like.

class `zenoh.`**`IValue`**

The IValue interface exposes how to recover a value's payload in a binary-serialized format, as well as that format's encoding.

`encoding`
The value's encoding

`payload`
The value itself, as an array of bytes

Python Module Index

Index