theia Documentation

Release 0.9.2

Pavle Jonoski

Mar 24, 2020

Contents:

CHAPTER 1

Installation Guide

Theia runs on python3, so before installing, please make sure that you have the latest version of python3 and pip (for python3). Use your package manager to install python3 and pip.

1.1 Installing with pip

To install with pip, just run:

```
pip install theia
```

Note that you may require `sudo` so it will install theia globally.

1.2 Installing from source

Prerequisites: you'll need `git` installed.

If you want to hack around in theia's code, you can install the source package.

You can install it with `pip`, or clone it with `git` in a virtual environment.

```
python -m venv dev
```

Then activate the environment:

```
source dev/bin/activate
```

1.2.1 Install with git

First cd to the directory where you want to checkout the source code, and clone the repository:

```
cd ~/projects/workspace
git clone https://github.com/theia-log/theia
```

Then cd to the theia directory and create the virtual environment:

```
cd theia
python -m venv ENV/dev
source ENV/dev/bin/activate
```

Then install the requirements:

```
pip install -r requirements.txt
```

1.2.2 Install with pip

First make sure you have your virtual environment set up and activated:

```
cd ~/projects/workspace
python -m venv ENV/dev
source ENV/dev/bin/activate
```

Go to the directory where you want to checkout the source code, install with pip from the Github repository:

```
pip install -e "git+https://github.com/theia-log/theia#egg=theia"
```

Confirm you've have the package properly installed:

```
python -m theia.cli --help
```

You should see the help message.

Then cd to the source directory in your ENV:

```
cd ENV/dev/src/theia
```

And you're ready to go.

CHAPTER 2

Developers Guide

2.1 Simulate Events

Theia comes with an events simulator script. This generates 'lorem-impsum' style events and pushes them to a running Theia collector server.

To run the script, make sure you've activated your virtual environment, then:

```
pip install lorem

python -m theia.cli.simulate -t tag1 tag2 tag3 -s src1 src2 src3 --context-size␣
↪120
```

This will run the event simulator that will generate events with random subset of the tags (tag1, tag2 and tag3) and random subset of sources (src1, src2 and src3) with a random content of size approximately 120 bytes (the size is randomized).

Full list of options for the simulator script:

- `-H HOST`, `--host HOST` - Collector host
- `-p PORT`, `--port PORT` - Collector port
- `-t [TAGS [TAGS ...]]` - Set of tags to choose from
- `-s [SOURCES [SOURCES ...]]` - Set of event sources to choose from
- `-c CONTENT` - Use this event content instead of random content.
- `--content-size CONTENT_SIZE` - Size of content (approximately)
- `--delay DELAY` - Delay between event in seconds

CHAPTER 3

Connector Websocket API

Theia collector is the main process that aggregates events from multiple sources. This is a WebSocket socket server that exposes endpoints for pushing and retrieving events. When retrieving events, the client must always supply a filter.

All exposed APIs are designed as event streams - i.e you can push event at any time reusing the same websocket without specifying the number of events that will be pushed. Similarly when retrieving events, you will receive the filter results without notification of the number of total matching events. The client will receive an event data message asynchronously - the only guarantee is that the event message itself is consistent (you won't receive a half message, either the full event passes through or none of it).

The collector exposes a real-time event stream as well that pushes events that match a certain filter criteria back to the client.

3.1 Event model

Each event is a textual message consisting of the following properties:

- `id` - the event ID. This value is unique across the whole system.
- `timestamp` - the time (and) date when the event was created. This is a UNIX-like timestamp, but may contain nanoseconds info.
- `source` - the source of the event. It may be the path of the log file being watched for changes, name of the sensor generating the event etc.
- `tags` - comma separated list of values acting as tags. Example: `web,httpd,access-log,server1, datacenter3`
- `content` - the actual content of the event. Plain text.

3.2 Event Format

The events are plain text strings. Here is an example of an event

```
id:331c531d-6eb4-4fb5-84d3-ea6937b01fdd
timestamp: 1509989630.6749051
source:/dev/sensors/door1-sensor
tags:sensors,home,doors,door1
Door has been unlocked.
```

The basic layout of an event looks like this

```
id:<id-string>
timestamp:<unix-timestamo>.<nanos>
source:<event-source>
tags:<comma-separated-tags>
<content>
```

The events are expected to be **UTF-8** encoded.

One event has two main parts: the header and the event content. The header has at least 4 lines: **id**, **timestamp**, **source** and **tags**. These are always present and always start with the line type then a colon **:**, then the header value (no empty spaces). If the header value is empty, then only the header name will be present plus the colon:

```
id:331c531d-6eb4-4fb5-84d3-ea6937b01fdd
timestamp: 1509989630.6749051
source:/dev/sensors/door1-sensor
tags:
Door has been unlocked.
```

Custom header fields may be added later on, but the current implementation supports and understands only the above four headers. The custom fields are not processed in any way, but are stored and returned to the clients. Currently the header lines order is not guaranteed to be preserved. Each header **MUST** be on a single line. It the client sends multiline value for a header, the collector will accept the value up until the line end and will ignore the rest.

The content starts after the last header and is separated by a single newline character.

3.3 Collector WebScoket API Endpoints

3.3.1 `/event` - Push (Add) Event

Open channel to push events.

- **Path**: `/event`
- **Params**: None
- **Message Payload**: Serialized Event string
- **Response**: None

Example of sending events to the collector (using wscat):

```
wscat -c ws://localhost:6433/event
connected (press CTRL+C to quit)
```

(continues on next page)

(continued from previous page)

```
> event: 110 108 2\nid:d55507cc-3530-47c1-913d-d07db6cfebea\ntimestamp: 1531528042.
↪9037790\nsource:/dev/sensors/temp0\ntags:sensor\n32\n
>
```

3.3.2 `/find` - Find events matching criteria

Endpoint params

- **Path**: `/find`
- **Params**: None
- **Message Payload**: first message body must be Criteria JSON.
- **Response**: Event stream

Opens channel to find events that match some criteria. The events are pushed from the collector to the client (on the incoming port of the web socket). The client should only post one message containing the filter criteria by which to match the events. This looks up only persisted events and will not return events in real-time, only those received **before** this channel was opened. Once all events have been send back to the client, the collector will close the websocket connection to the client.

The criteria message must a JSON string with the following format:

```
{
    "start": int,
    "end": int,
    "tags": ["string regex", ...],
    "content": "string regex",
    "order": "asc|desc"
}
```

Where:

- `start` - `int`, *optional*: match events **after** this timestamp (UNIX).
- `end` - `int`, *optional*: match events **before** this timestamp (UNIX).
- `tags` - array of `string`, *optional*: match the events matching any of the supplied tags. The values are processed as regular expressions.
- `content` - `string` regular expression, *optional*: match the eventa with content matching to the supplied content regex.
- `order` - `string` one of `asc` or `desc`, *optinal*: sort order for the result. The sort is performed by the event timestamp. By default it returns the events in ascending order (`asc`) which means earlier events are returned first.

Example

Match all events after a timestamp that have a tag `log` on any `web-server` and contain `[ERROR]`:

```
{
    "start": 1527283299,
    "tags": ["log", "web-server-.+"],
    "content": ".*\[ERROR\].*"
}
```

Example (assuming the collector runs on localhost):

```
wscat -c ws://localhost:6433/find
connected (press CTRL+C to quit)
> {"start": 1531528038}
< ok
< event: 110 108 2
id:2ca00a2a-d70f-4617-b48f-a31716b1d5dc
timestamp: 1531528038.8951149
source:/dev/sensors/temp0
tags:sensor
32
```

Notice that the first result is the string `ok` - this means that the query was successfully processed by the server. The next messages are the matched events. Every serialized event message always ends in a newline.

3.3.3 `/live` - Real-time event stream

Endpoint params

- **Path**: `/live`
- **Params**: None
- **Message Payload**: first message body must be Criteria JSON.
- **Response**: Event stream

Opens channel to monitor for events matching a certain criteria. The client can open a channel to the collector to monitor for incoming events that match the client criteria. This endpoint will **not** lookup events in the persistent storage, but matches only the events coming to the collector **after** the channel was opened.

The collector does not close this channel. If a timeout occurs due to inactivity, then the client must initiate new websocket connection.

The first message sent to the collector after establishing the channel **must** be the filter criteria object serialized as JSON string.

The criteria object has the following format:

```
{
    "id": "string regex",
    "start": int,
    "end": int,
    "tags": ["string regex", ...],
    "source": "string regex",
    "content": "string regex"
}
```

Where:

- `id` - `string` regular expression, *optional*: match any event which `id` matches the provided regular expression.
- `start` - `int`, *optional*: match events **after** this timestamp (UNIX).
- `end` - `int`, *optional*: match events **before** this timestamp (UNIX).
- `tags` - array of `string`, *optional*: match the events matching any of the supplied tags. The values are processed as regular expressions.
- `source` - `string` regular expression, *optional*: match any event which `source` matches the provided regular expression.

- `content` - `string` regular expression, *optional*: match the eventa with content matching to the supplied content regex.

Example

Match all events after a timestamp that have a tag `log` on any `web-server` and contain `[ERROR]` from the `/var/log` files (source):

```
{
    "start": 1527283299,
    "tags": ["log", "web-server-.+"],
    "content": ".*\[ERROR\].*",
    "source": "/var/log/.+"
}
```

Example using `wscat` (assuming the collector runs on localhost):

```
wscat -c ws://localhost:6433/live
connected (press CTRL+C to quit)
> {"start": 1531528038}
< ok
< event: 110 108 2
id:2ca00a2a-d70f-4617-b48f-a31716b1d5dc
timestamp: 1531528038.8951149
source:/dev/sensors/temp0
tags:sensor
32
```

3.4 Simple event parser and serializer in JavaScript

An event parser and serialized in JavaScript.

```
function parseEvent(event_str) {
    let event = {}

    let lines = event_str.split('\n')

    for (var i = 0; i < lines.length; i++) {
        let line = lines[i]
        let idx = line.indexOf(':')
        if (idx < 0) {
            break
        }
        let prop = line.slice(0, idx);
        let value = line.slice(idx+1, line.length);

        if (prop == 'tags') {
            value = value.split(',').filter( t => { return t; });
        }

        event[prop] = value
    }
    if (i < lines.length) {
        event.content = lines.slice(i, lines.length).join('\n')
    }

```

(continues on next page)

(continued from previous page)

```
    return event
}

function serializeEvent(event) {
    let event_str = ''
    let guaranteed = ['id', 'timestamp', 'source', 'tags']
    for (var i = 0; prop = guaranteed[i]; i++) {
        let value = event[prop];
        if (prop == 'tags') {
            value = value.join(',');
        }
        event_str += prop + ':' + value + '\n';
    }

    for (var prop in event) {   // add custom headers
        if (!guaranteed.includes(prop) && prop != 'content') {
            event_str += prop + ':' + event[prop] + '\n';
        }
    }

    event_str += event.content;
    return event_str
}


var event_str = ['id:331c531d-6eb4-4fb5-84d3-ea6937b01fdd',
                 'timestamp: 1509989630.6749051',
                 'source:/dev/sensors/door1-sensor',
                 'tags:sensors,home,doors,door1',
                 'x-header:somevalue',
                 'Door has been unlocked.'].join('\n')

var event = parseEvent(event_str);
console.log(event)
// prints:
// { id: '331c531d-6eb4-4fb5-84d3-ea6937b01fdd',
//   timestamp: ' 1509989630.6749051',
//   source: '/dev/sensors/door1-sensor',
//   tags: [ 'sensors', 'home', 'doors', 'door1' ],
//   'x-header': 'somevalue',
//   content: 'Door has been unlocked.' }


var serialized = serializeEvent(event);
console.log(serialized);
// prints:
// id:331c531d-6eb4-4fb5-84d3-ea6937b01fdd
// timestamp: 1509989630.6749051
// source:/dev/sensors/door1-sensor
// tags:sensors,home,doors,door1
// x-header:somevalue
// Door has been unlocked.
```

CHAPTER 4

Theia API Docs

Full API docs

4.1 theia.collector

The log aggregator collector server implementation.

class `theia.collector.`**`Collector`**(*store*, *hostname='0.0.0.0'*, *port=4300*, *persistent=True*)

Collector server.

Collects the events, passes them down the live pipe filters and stores them in the event store.

Parameters

- **`store`** – `theia.storeapi.Store`, store instance
- **`hostame`** – `str`, server hostname. Default is '0.0.0.0'.
- **`port`** – `int`, server port. Default is 4300.

`run`()

Run the collector server.

This operation is blocking.

`stop`()

Stop the collector server.

This operation is non blocking.

class `theia.collector.`**`Live`**(*serializer*)

Live event pipeline.

Each event is passed through the live event pipeline and matched to the LiveFilter filters.

Parameters **`serializer`** – `theia.model.Serializer`, event serializer.

add_error_handler(*handler*)

Adds error handler. The handler will be called for each error that occurs while processing the filters in this live pipe.

Parameters **handler** (*function*) – error handler callback. The handler has the following prototype:

```
def handler(err, websocket, live_filter):
    pass
```

where:

- `err` (`Exception`) the actual error.
- `websocket` (`websockets.WebSocketClientProtocol`) reference to the WebSocket instance.
- `live_filter` (*LiveFilter*) filter criteria.

add_filter(*lfilter*)

Adds new filter to the pipeline.

Parameters **lfilter** – *LiveFilter*, the filter to add to the pipeline.

pipe(*event*)

Add an event to the live pipeline.

The event will be matched against all filters in this pipeline.

Parameters **event** – *theia.model.Event*, the event to be pipelined.

class theia.collector.**LiveFilter**(*ws*, *criteria*)

Filter for the live event pipe.

Holds a single criteria to filter events by.

Parameters

- **ws** – `websockets.WebSocketClientProtocol`, reference to the web socket instance.
- **criteria** – `dict`, dict holding criteria values.

match(*event*)

Matches an event to the criteria of this filter.

Parameters **event** – *theia.model.Event*, the event to match.

Returns match(`bool`), True if the event matches the criteria, otherwise False.

4.2 theia.comm

Theia communication module.

Defines classes for building theia servers and clients.

Theia communication module is asynchronous and is built on top of asyncio loop.

There are two main interfaces:

- `Server` an async server handling and managing WebSocket connections from multiple clients.
- `Client` an async client connection to a theia server.

The interfaces are designed to work primarily with theia events and are thread-safe.

class `theia.comm.`**`Client`**(*loop*, *host*, *port*, *secure=False*, *path=None*, *recv=None*)

Client represents a client connection to a theia server.

Parameters

- **`loop`** – `asyncio` EventLoop to use for this client.
- **`host`** – `str`, theia server hostname.
- **`port`** – `int`, theia server port.
- **`secure`** – `bool`, is the connection secure.
- **`path`** – `str`, the request path - for example: `"/live"`, `"/events"` etc.
- **`recv`** – `function`, receive handler. Called when a message is received from the server. The handler has the following signature:

```
def handler(message):
    pass
```

where:

- `message` is the message received from the theia server.

`close`(*reason=None*)

Close the connection to the remote server.

Parameters **`reason`** – `str`, the reason for disconnecting. If not given, a default `"normal close"` is sent to the server.

`connect`()

Connect to the remote server.

`is_open`()

Check if the client connection is open.

Returns `True` if the client connection is open, otherwise `False`.

`on_close`(*handler*)

Add close handler.

The handles is called when the client connection is closed either by the client or by the server.

Parameters **`handler`** – `function`, the handler callback. The callback prototype looks like so:

```
def callback(websocket, code, reason):
    pass
```

where:

- **`websocket`** **`websockets.WebSocketClientProtocol`** **is the underlying** websocket.
- **`code`** **`int`** **is the code received when the connection was closed. Check** out the WebSocket specification for the list of codes and their meaning.
- `reason` `str` is the reason for closing the connection.

`send`(*message*)

Send a `str` message to the remote server.

Parameters **`message`** – `str`, the message to be sent to the remote server.

Returns the `asyncio.Handle` to the scheduled task for sending the actual data.

`send_event`(*event*)

Send an event to the remote server.

Serializes, then sends the serialized content to the remote server.

Parameters **`event`** – *`theia.model.Event`*, the event to be send.

Returns the `asyncio.Handle` to the scheduled task for sending the actual data.

`class` `theia.comm.`**`Server`**(*loop*, *host='localhost'*, *port=4479*)

Listens for and manages multiple client connections.

The server is based on `asyncio` event loop. It manages the websocket connections comming from multiple clients based on the path in the websocket request connection.

Provides a way to register a callback for notifying when a client connects to a particular endpoint (path), and also a way to register a callback for when the client disconnects.

Instances of this class are thread-safe.

Parameters

- **`loop`** – `asyncio.BaseEventLoop`, the event loop.
- **`host`** – `str`, the hostname to bind to when listening fo incoming connections.
- **`port`** – `int`, the port to listen on.

`on_action`(*path*, *cb*)

Register a callback to listen for messages from clients that connected to this specific entrypoint (path).

The callback will be called whenever a new message is received from the client on this `path`.

If multiple callbacks are registered on the same action, then they are called one by one in the same order as registered. The response from the callbacks is chained between the subsequent calls.

Parameters

- **`path`** – `str`, the request path of the incoming websocket connection.
- **`cb`** – `function`, the callback to be called when a message is received from the client on this path. The callback handler looks like this:

```
def callback(path, message, websocket, resp):
    return resp
```

where:

- `path` `str`, the path on which the message was received.
- `message` `str`, the messge received from the websocket connection.
- **`websocket` `websockets.WebSocketClientProtocol`, the underlying** websocket.
- **`resp` `str`, the response from the previous action registered on this** same action

The callback must return `str` response or `None`.

`on_websocket_close`(*websocket*, *cb*)

Register a close callback for this websocket.

Parameters

- **`websocket`** – `websockets.WebSocketClientProtocol`, the websocket to watch for closing.

- **cb** – `function`, the callback to be called when the `websocket` is closed. The callback should look like this:

```
def callback(ws, path):
    pass
```

where:

- **ws (`websockets.WebSocketClientProtocol`), the underlying websocket** connection.
- `path` (`str`), the request path of the websocket.

This method returns `True` if the callback was added; `False` if the websocket is not managed by this *`Server`* instance.

`start`()

Starts the server.

This call blocks until the server is started or an error occurs.

`stop`()

Stops the server.

Closes all client websocket connections then shuts down the server.

This operation blocks until the server stops or an error occurs.

class `theia.comm.`**`wsHandler`**(*websocket*, *path*)

Wrapper for an incoming websocket connection.

Used primarily with the *`Server`* implementation in the client connections life-cycle management.

Parameters

- **websocket** – `websockets.WebSocketClientProtocol`, the underlying websocket connection.
- **path** – `str`, the request path of the websocket connection.

Note: This class is mainly used internally and as such it is a subject of chnages in its API.

`add_close_handler`(*hnd*)

Register a close handler for this connection.

Parameters **hnd** – `function`, the close handler callback. The callback receives two parameters:

- **ws (`websockets.WebSocketClientProtocol`), the underlying websocket** connection.
- `path` (`str`), the request path of the websocket.

`trigger`(*websocket*)

Triggers the close handlers for this websocket.

Parameters **websocket** – `websockets.WebSocketClientProtocol`, the underlying websocket connection.

4.3 theia.model

Theia event data model.

Basic model of an Event, serializers and parsers for Event manipulation.

exception theia.model.**EOFException**

Represents an error in parsing an *Event* from an underlyign stream that occurs when the end of stream is reached prematurely.

class theia.model.**Event**(*id*, *source*, *timestamp=None*, *tags=None*, *content=None*)

Event represnts some event occuring at a specfic time.

Each event is uniquely identified by its `id` in the whole system. An event comes from a `source` and always has an associated `timestamp`. The timestamp is usually generated by the event producer.

The *content* of an event is an arbitrary string. It may be a log file line, some generated record, readings from a sensor or other non-structured or structured text.

Each event may have a list of `tags` assocciated with it. These are arbitrary strings and help in filtering the events.

An event may look like this

```
id:331c531d-6eb4-4fb5-84d3-ea6937b01fdd
timestamp: 1509989630.6749051
source:/dev/sensors/door1-sensor
tags:sensors,home,doors,door1
Door has been unlocked.
```

The constructor takes multiple arguments, of which only the id and source are required.

Parameters

- **id** – `str`, the event unique identifier. Must be system-wide unique. An UUID version 4 (random UUID) would be a good choice for `id`.
- **source** – `str`, the source of the event. It usually is the name of the monitored file, but if the event does not originate from a file, it should be set to the name of the process, system or entity that generated the event.
- **timestamp** – `float`, time when the event occured in seconds (like UNIX time). The value is a floating point number with nanoseconds precission. If no value is given, then the current time will be used.
- **tags** – `list`, list of `str` tags to add to this event.
- **content** – `str`, the actual content of the event. The content may have an arbitrary lenght (or none at all).

match(*id=None*, *source=None*, *start=None*, *end=None*, *content=None*, *tags=None*)

Check if this event matches the provided criteria.

The event will match only if **all** criteria is statisfied. Calling match without any criteria, yields `True`.

The criteria is processed as a regular expression. Each value is first converted to string, then matched against the provided regular expression - see `re.match()`. The exception of this rule are the criteria for `start` and `end` wich expect numeric values, as they operate on the *Event* timestamp.

Parameters

- **id** – `str`, regular expression against which to match the *Event* `id`.
- **source** – `str`, regular expression against which to match the *Event* `source`.
- **start** – `float` or `int`, match true if the *Event* timestamp is greater than or equal to this value.
- **start** – `float` or `int`, match true if the *Event* timestamp is less than or equal to this value.

- **content** – `str`, regular expression against which to match the *`Event`* `content`.
- **tags** – `list`, list of `str` regular expressions against which to match the *`Event`* tags. Matches true only if **all** of the provided criteria match the Event tags.

Returns `True` if this *`Event`* matches the criteria, otherwise `False`.

class `theia.model.`**`EventParser`**(*encoding='utf-8'*)

Parses an incoming bytes stream into an *`Event`*.

Offers methods for parsing parts of an *`Event`* or parsing the full event from the incoming `io.BytesIO` stream.

The stream will be decoded before converting it to `str`. By default the parser assumes that the stream is `utf-8` encoded.

Parameters **encoding** – `str`, the encoding to be ued for decoding the stream bytes. The default is `utf-8`.

`parse_event`(*stream*, *skip_content=False*)

Parses an event from the incoming stream.

The parsing is done in two phases:

1. **The preamble is parsed to determine the total size of the event and the** the size of the event header.
2. Then the actual event is read, either with or without the event content.

If `skip_content` is set to `True`, then the actual content of the event is not read. This is usefull in event readers that do matching of the event header values, without wasting memory and performance for reading the content. In this case, the event content will be set to `None`.

Parameters

- **stream** – `io.BytesIO`, the stream to parse from.
- **skip_content** – `bool`, whether to skip the fetching of the content and to fetch only the *`Event`* header. Default is `False`.

Returns the parsed *`Event`* from the incoming stream.

`parse_header`(*hdr_size*, *stream*)

Parses the *`Event`* header into a *`Header`* object from the incoming stream.

First `hdr_size` bytes are read from the `io.BytesIO` stream and are decoded to `str`.

Then, the parser parses each line by splitting it by the first colon (**`:`**). The first part is ued to determine the *`Header`* property. The part after the colon is the propery value.

Parameters

- **hdr_size** – `int`, the size of the header in bytes. See *`EventParser.parse_preamble()`* on how to determine the header size in bytes.
- **stream** – `io.BytesIO`, the incoming stream to parse.

Returns the parsed *`Header`* for the event.

Raises `Exception` if an unknown property is encountered in the header.

`parse_preamble`(*stream*)

Parses the event preamble from the incoming stream into a *`EventPreamble`*.

The event preamble is a single line read from the stream with the following structure:

```
<total_size> <header_size> <content_size>
```

where:

- `total_size` is the total size of the Event (after the heading) in bytes.
- `header_size` is the size of the header in bytes.
- `content_size` is the size of the content (after the Header) in bytes.

Note that the sizes are expressed in bytes.

Parameters **`stream`** – `io.BytesIO`, the stream to parse.

Returns the parsed *`EventPreamble`* from the incoming stream.

class `theia.model.`**`EventPreamble`**(*total*, *header*, *content*)

A preamble to an *`Event`*.

The preamble is present only in the serialized representation of an event and holds the information about the size of the event and its parts.

`content`

`int`, the size of the serialized event content in bytes.

`header`

`int`, the size of the serialized event header in bytes.

`total`

`int`, the size of the serialized event in bytes.

class `theia.model.`**`EventSerializer`**(*encoding='utf-8'*)

Serialized for instances of type *`Event`*.

This serializes the *`Event`* in a plain text representation of the Event. The serialized text is encoded in UTF-8 and the actual `bytes` are returned.

The representation consists of three parts: preamble, header and content. The preamble is the first line of every event and has this format:

event: <total_size> <header_size> <content_size>

where:

- `total_size` is the total size of the Event (after the heading) in bytes.
- `header_size` is the size of the header in bytes.
- `content_size` is the size of the content (after the Header) in bytes.

The header holds the values for the Event's id, source, tags and timestamp. Each value is serialized on a single line. The line starts with the name of the property, separated by a colon(`:`), then the property value.

The content starts after the final header and is separated by a newline.

Here is an example of a fully serialized *`Event`* (Python's `bytes`):

```
b'event: 155 133 22\nid:331c531d-6eb4-4fb5-84d3-ea6937b01fdd\ntimestamp:␣
↪1509989630.6749051\nsource:/dev/sensors/door1-sensor\ntags:sensors,home,doors,
↪door1\nDoor has been unlocked\n'
```

or as a textual representation:

```
event: 155 133 22
id:331c531d-6eb4-4fb5-84d3-ea6937b01fdd
timestamp: 1509989630.6749051
source:/dev/sensors/door1-sensor
tags:sensors,home,doors,door1
Door has been unlocked
```

Note that the *EventSerializer* adds a trailing newline (`\n`) at the end.

The serializer constructor takes the encoding as an argument. By default "utf-8" is used.

Parameters **encoding** – `str`, the encoding of the serialized string. Default `utf-8`.

serialize (*event*)

Serializes an *Event*.

See *EventSerializer* for details on the serialization format.

Parameters **event** – *Event*, the event to be serialized.

Returns the serialized event as `bytes`.

class theia.model.**Header** (*id=None*, *timestamp=None*, *source=None*, *tags=None*)

Header represents an Event header. The header contains the following properties:

- `id`, unique identifier for the event. Usually UUIDv4.
- `timestamp`, floating point of the number of milliseconds since epoch start (1970-1-1T00:00:00.00).
- `source`, string, the name of the event source.
- `tags`, list of strings, arbitrary tags attached to the event.

The header is usefull and usually used when serializing/parsing an event.

4.4 theia.naivestore

Naive implementation of the Event Store.

This module provides an implementation of the *theia.storeapi.EventStore* that stores the events in plain-text files.

The store writes to the files atomically, so there is no danger of leaving the files in an inconsistent state.

The files in which the store keeps the events are plain text files that contain serialized events, encoded in UTF-8. The events are written sequentially. Plain text is chosen so that these files can be also be read and processed by other tools (such as grep). The events are kept in multiple files. Each file contains about a minute worth of events - all events that happened in that one minute time span. The name of the file is the time span: <first-event-timestamp>-<last-event-timestamp>.

The naive store requires a root directory in which to store the events. Here is an example of usage of the store:

```
from theia.naivestore import NaiveEventStore
from theia.model import Event
from uuid import uuid4
from datetime import datetime

store = NaiveEventStore(root_dir='./data')

timestamp = datetime.now().timestamp()
```

(continues on next page)

(continued from previous page)

```
store.save(Event(id=uuid4(),
                 source='test-example',
                 timestamp=timestamp,
                 tags=['example'],
                 content='event 1'))
store.save(Event(id=uuid4(),
                 source='test-example',
                 timestamp=timestamp + 10,
                 tags=['example'],
                 content='event 2'))
store.save(Event(id=uuid4(),
                 source='test-example',
                 timestamp=timestamp + 20,
                 tags=['example'],
                 content='event 3'))

# now let's search some events

for ev in store.search(ts_start=timestamp + 5):
    print('Found:', ev.content)
```

would print:

```
>> Found: event 2
>> Found: event 3
```

class theia.naivestore.**DataFile**(*path*, *start*, *end*)

Represents a file containing data (events) within a given time interval (from `start` to `end`).

end

`int`, timestamp, the end of the interval. The data file does not contain any events after this timestamp.

path

`str`, the path to the data file.

start

`int`, timestamp, the start of the interval. The data file does not contain any events before this timestamp.

class theia.naivestore.**FileIndex**(*root_dir*)

An index of *`DataFile`* files loaded from the given directory.

Loads an builds an index of *`DataFile`* files from the given directory. Each data file name must be in the form: `<start>-<end>`, where `start` and `end` represent the time interval of the events in that data file.

`FileIndex` loads all data files and builds a total time span of all data files. The index exposes methods for locating and searching files that contain the events within a given time interval.

Parameters **root_dir** – `str`, the directory from which to load the data files.

add_file(*fname*)

Add a data file to the *`FileIndex`*.

The time span will be recalculated to incorporate this new data file.

Parameters **fname** – `str`, the file name of the data file to be added to the file index.

find(*ts_from*, *ts_to*)

Finds the data files that contain the events within the given interval [`ts_from`, `ts_to`].

The interval can be open at the end ([ts_from, Inf]), by passing 0 for ts_to. The interval cannot be open at the start.

Parameters

- **ts_from** – float, timestamp, find all data files containing events that have timestamp greater than or equal to ts_from.
- **ts_to** – float, timestamp, find all data files containing events that have timestamp less than or equal to ts_to. If 0 or None is passed for this parameter, then the end of the time span is open, meaning this parameter will be ignore in the search.

Returns a list of *DataFile* files that contain the events within the given interval. Returns None if there are no files containing events within the given interval.

find_event_file(*timestamp*)
Find the event file that contains the event with the given timestamp.

Parameters **timestamp** – int, the timestamp of the event.

Returns the *DataFile* containing the event, or None if it cannot be found.

class theia.naivestore.**MemoryFile**(*name*, *path*)
File-system backed in-memory buffer.

This class wraps an io.BytesIO buffer and backs it up with a real file in the file-system. The writes go to the in-memory buffer, which then can be flushed to the actual file in the file-system.

The flushing of the buffer is atomic and consistent. The buffer is first flushed to a temporary file, then the system buffers are synced, and then the temporary file is renamed as the actual file.

The instances of this class are thread-safe and can be shared between threads.

Limitations: This class is not optimized for large data files as it keeps all of the file data in memory. This may cause a performance penalty in both speed and memory consumption when dealing with large files. In those cases it is better to use other memory mapped file implementations.

Parameters

- **name** – str, the name of the file in the file-system. This is just the filename, without the directory.
- **path** – str, the directory holding the file in the file-system.

flush()
Writes the in-memory buffer to the file in the file-system.

This operation is atomic and guarantees that the complete state of the buffer will be written to the file. The underlying file will never be left in an inconsistent state. This is achieved by first writing the entire buffer to a temporary file (in the same directory), flushing the system buffers, then if this succeeds, renaming the temporary file as the original file name.

stream()
Returns a copy of the underlying io.BytesIO in-memory stream.

write(*data*)
Write to the data to the buffer.

This writes the data to the in-memory buffer.

Parameters **data** – bytes, the data to be written to the buffer.

class theia.naivestore.**NaiveEventStore**(*root_dir*, *flush_interval=1000*)
A naive implementation of the *theia.storeapi.EventStore* that keeps the event data is a plain text files.

The events are kept in plain text files, serialized by default in UTF-8. The files are human readable and the format is designed to be (relatively) easily processed by other tools as well (such as `grep`). Each data file contains events that happened within one minute (60000ms). The names of the data files also reflect the time span interval, so for example a file with name *1528631988-1528632048* contains only events that happened at or after `1528631988`, but before `1528632048`.

The store by default uses in-memory buffers to write new events, and flushes the buffer periodically. By default the flushing occurs roughly every second (1000ms, see the parameter `flush_interval`). This increases the performance of the store, but if outage occurs within this interval, the data in the in-memory buffers will be lost. The store can be configured to flush the events immediately on disk (by passing `0` for `flush_interval`), but this decreases the performance of the store significantly.

The instances of this class are thread-safe and can be shared between threads.

Parameters

- **`root_dir`** – `str`, the root directory where to store the events data files.
- **`flush_interval`** – `int`, flush interval for the data files buffers in milliseconds. The event data files will be flushed and persisted on disk every `flush_interval` milliseconds. The default value is 1000ms. To flush immediately (no buffering), set this value equal or less than `0`.

`close`()

Close and cleanup the underlying store.

`delete`(*event_id*)

`NaiveEventStore` does not support indexing, so a delete by `id` is not supported.

`get`(*event_id*)

`NaiveEventStore` does not support indexing, so search by `id` is also not supported.

`save`(*event*)

Saves an event in the underlying storage.

This method is guaranteed to be atomic in the sense that the storage will either succeed to write and flush the event, or it will fail completely. In either case, the storage will be left in a consistent state.

Parameters **`event`** – *`theia.model.Event`*, the Event object to store.

This method does not return any value.

`search`(*ts_start*, *ts_end=None*, *flags=None*, *match=None*, *order='asc'*)

Performs a search for events matching events in the specified time range.

Parameters

- **`ts_start`** – `float`, start of the time range. Matching events with timestamp bigger or equal to this parameter will be returned.
- **`ts_end`** – `float`, end of the time range. Matching events with timestamp smaller or equal to this parameter will be returned.
- **`flags`** – `list`, events that have ALL of the flags will be returned.
- **`match`** – `str`, regular expression, (restricted to a subset of the full regexp support) to match the event content against.
- **`order`** – `str`, `'asc'` or `'desc'`, order in which the events are returned.

The operation returns an iterator over the matched (ordered) set of events. This operation satisfies the strict consistency.

class `theia.naivestore.`**`PeriodicTimer`**(*interval*, *action*)

Timer that executes an action periodically with a given interval.

The timer executes the action in a separate thread (as this class is a subclass of `threading.Thread`). To run the action you must call `PeriodicTimer.start()`. The first execution of the action is delayed by `interval` seconds. This timer does not call the `action` callback every `interval` seconds, but rather waits `interval` seconds between after the action completes until the next call. So for a long running tasks, the time of call of the action may not be evenly spaced.

You can cancel this timer by calling meth:*PeriodicTimer.cancel*.

Parameters

- **`interval`** – `numeric`, seconds to wait between subsequent calls to `action` callback.
- **`action`** – `function`, the action callback. This callback takes no arguments.

`cancel`()

Cancels the running timer.

The timer thread may continue running until the next cycle, then it exits.

`run`()

Runs the periodic timer.

Do **not** call this function directly, but rather call `PeriodicTimer.start()` to run this thread.

To cancel the timer, call *`PeriodicTimer.cancel()`*.

class `theia.naivestore.`**`SequentialEventReader`**(*stream*, *event_parser*)

Reads events (*`theia.model.Event`*) from an incoming `io.BytesIO` stream.

Uses an *`theia.model.EventParser`* to parse the events from the incoming stream.

Provides two ways of parsing the events:

- Parsing the event fully - loads the header and the content of the event. See *`SequentialEventReader.events()`*.
- **Parsing only the event header - this skips the loading of the content. Useful for not wasting performance/memory**
  on loading and decoding the event content when not searching by the event content.

This reader implements the context manager interface and can be used in `with` statements. For example:

```
with SequentialEventReader(stream, parser) as reader:
    for event in reader.events():
        print(event)
```

Parameters

- **`stream`** – `io.BytesIO`, the incoming stream to read events from.
- **`event_parser`** – *`theia.model.EventParser`*, the parser used for parsing the events from the stream.

`curr_event`()

Reads an *`theia.model.Event`* at the current position in the stream.

Reads the event fully.

Returns the *`theia.model.Event`* at the current position of the stream or `None` if there are no more available events to be read from the stream (the stream closes).

events()

Reads full events from the incoming stream.

Returns an iterator for the read events and yields *theia.model.Event* as it becomes available in the stream.

The iterator stops if there are no more events available in the stream or the stream closes.

events_no_content()

Reads events without content (just header) from the incoming stream.

Returns an iterator for the read events and yields *theia.model.Event* as it becomes available in the stream.

Note that the `content` property of the *theia.model.Event* will always be set to None.

The iterator stops if there are no more events available in the stream or the stream closes.

theia.naivestore.**binary_search**(*datafiles*, *timestamp*)

Performs a binary search in the list of datafiles, for the index of the first data file that contain events that happened at or after the provided timestamp.

Parameters

- **datafiles** – `list` of *DataFile*, list of datafiles **sorted** in ascending order by the start timestamp.
- **timestamp** – `float`, the timestamp to serch for.

Returns the index (`int`) of the first *DataFile* in the list that contains event that occurred at or after the provided timestamp. Returns `None` if no such data file can be found.

4.5 theia.query

Query theia server for events.

This module contains the *Query* that implements API for querying theia collector server.

class theia.query.**Query**(*host*, *port*, *secure=False*, *loop=None*)

Represents a query to a theia collector server.

The query instances are thread-safe.

Parameters

- **host** – `str`, the hostname (IP) of the collector server.
- **port** – `int`, the port of the collector server.
- **secure** – `bool`, whether to connect securely to the collector server.
- **loop** – `asyncio.BaseEventLoop`, the event loop to use.

find(*criteria*, *callback=None*)

Make a query to the collector.

The collector will search for any events that occured **before** theis query was sent to the server and will return those that match the given `criteria`.

Parameters

- **criteria** – `dict`, the criteria filter for the events. This is a `dict` that can contain the following keys:

  - `id`, `str`, pattern for matching the event id.
  - `source`, `str`, pattern for matching the event source.
  - `start`, `int`, match events with timestamp greater or equal to this.
  - `end`, `int`, match events with timestamp less than or equal to this.
  - `content`, `str`, pattern for matching the event content.
  - `tags`, `list`, list of patterns to match against the event tags.
- **`callback`** – `function`, called with the matching event content. The method takes one argument, the serialized *`theia.model.Event`*.

Returns a *`ResultHandler`*.

`live`(*criteria*, *callback=None*)
Make a live query to the collector.

The collector will watch for any events that occur **after** the live query is registered and return those that match the given `criteria`.

Parameters

- **`criteria`** – `dict`, the criteria filter for the events. This is a `dict` that can contain the following keys:
  - `id`, `str`, pattern for matching the event id.
  - `source`, `str`, pattern for matching the event source.
  - `start`, `int`, match events with timestamp greater or equal to this.
  - `end`, `int`, match events with timestamp less than or equal to this.
  - `content`, `str`, pattern for matching the event content.
  - `tags`, `list`, list of patterns to match against the event tags.
- **`callback`** – `function`, called with the matching event content. The method takes one argument, the serialized *`theia.model.Event`*.

Returns a *`ResultHandler`*.

`class` `theia.query.`**`ResultHandler`**(*client*)
Represents a result of a query against a theia collector server.

The result of a query to the collector server is a stream of events. A callback (see `callback` in *`Query.live()`*) can be set to be called whenever an *`theia.model.Event`* is received, but the control to close the stream is passed down to a *`ResultHandler`*.

This class wraps the *`theia.comm.Client`* used to connect to the collector server and adds support for registering on client closed events. It also adds means to close (cancel) the connection to the theia server.

Parameters **`client`** – *`theia.comm.Client`*, the underlying client to the theia server.

`cancel`()
Cancel this result.

Closes the underlying client connection.

`when_closed`(*closed_handler*)
Register a handler to be called when the connection to the server is closed.

The handler has the following signature:

```
def closed_handler(client, code, reason):
    pass
```

where:

- `client` - *`theia.comm.Client`*, is the underlying client connected to the theia collector.
- `code` - `int`, wbsocket close connection code.
- `reason` - `str`, the reason for closing the connection.

4.6 theia.storeapi

Event Store API

Defines classes, methods and exceptions to be used when implementing an Event Store.

exception `theia.storeapi.`**EventNotFound**

Raised if there is no event found in the underlying storage.

exception `theia.storeapi.`**EventReadException**

Represents an error while reading an event from the underlying storage.

class `theia.storeapi.`**EventStore**

EventStore is the basic interface for interaction with the events.

Main uses of this store are CRUD interactions with the events. The API provides powerful search through all events based on a time range and optionally additional flags. An instance of this class is thread-safe.

close()

Close and cleanup the underlying store.

delete(*event_id*)

Deletes an event from the storage.

The delete operation removes an event from the underlying storage. This operation is guaranteed to be atomic, the event will either be removed or it will fail completely. In either case the storage will be left in a consistent state.

Parameters **event_id** – `str`, the unique identifier of the event to be removed.

This method does not return any value.

get(*event_id*)

Looks up an event by its unique identifier.

The storage will try to look up the event with the specified id:

- if the event is found, it will return an Event object
- the event is not found, raises an EventNotFound exception.

Edge cases:

- **If the event is being inserted AFTER the get(..) operation is invoked,** there is NO guarantee that it will be fetched.
- **If the event is being inserted BEFORE the get(..) operation is invoked,** but that transaction is still not committed, the operation will block until the write operation completes (or errors out) and the Event will be returned (if the write succeeds) or will error out (if the write fails) - strict consistency

Note: Some specific implementations may break the strict consistency if the underlying mechanism does not provide means to implement it. In those cases, the subclass must override this documentation and must document its exact for the above edge cases.

Parameters **`event_id`** – `str`, the unique identifier of the event to be looked up.

Returns the *`theia.model.Event`* with the given id, or `None` if no such event exists.

`save`(*event*)

Saves an event in the underlying storage.

This method is guaranteed to be atomic in the sense that the storage will either succeed to write and flush the event, or it will fail completely. In either case, the storage will be left in a consistent state.

Parameters **`event`** – *`theia.model.Event`*, the Event object to store.

This method does not return any value.

`search`(*ts_start*, *ts_end=None*, *flags=None*, *match=None*, *order='asc'*)

Performs a search for events matching events in the specified time range.

Parameters

- **`ts_start`** – `float`, start of the time range. Matching events with timestamp bigger or equal to this parameter will be returned.
- **`ts_end`** – `float`, end of the time range. Matching events with timestamp smaller or equal to this parameter will be returned.
- **`flags`** – `list`, events that have ALL of the flags will be returned.
- **`match`** – `str`, regular expression, (restricted to a subset of the full regexp support) to match the event content against.
- **`order`** – `str`, `'asc'` or `'desc'`, order in which the events are returned.

The operation returns an iterator over the matched (ordered) set of events. This operation satisfies the strict consistency.

exception `theia.storeapi.`**`EventStoreException`**

General store error.

exception `theia.storeapi.`**`EventWriteException`**

Represents an error while writing an event to the underlying storage.

4.7 theia.rdbs

Relational database EventStore implementation.

class `theia.rdbs.`**`EventRecord`**(***kwargs*)

SQLAlchemy model representing the event.

class `theia.rdbs.`**`RDBSEventStore`**(*session_factory*)

EventStore that persists the events in a relational database.

The implementation relies on SQLAlchemy ORM framework.

Parameters **`session_factory`** – the SQLAlchemy SessionMaker function.

close()
Closes the store.

Does nothing in this implementation.

delete(*event_id*)
Deletes the event with the given event id.

Parameters **event_id(str)** – the event id

get(*event_id*)
Looks up an event by its id.

Parameters **event_id(str)** – the event id.

Returns a `theia.model.Event` if the event was found or `` `theia.storeapi.EventNotFound` `` if no such event can be found.

save(*event*)
Stores the event in the underlying database.

Note that this method only does *INSERT* as the `EventStore` has no concept of *UPDATE* - each event is only added and cannot be updated. Adding the same event twice will result in an error.

Parameters **(theia.model.Event)** (`event`) – the event to save.

This method does not return a value.

search(*ts_start*, *ts_end=None*, *flags=None*, *match=None*, *order='asc'*)
Performs a search through the stored events.

Parameters

- **ts_start(float)** – *required*, match all events that occured at or later than this time.
- **ts_end(float)** – *optional*, match all events that occured before this time.
- **flags(list)** – *optional*, list of string values (regular expressions) against which to match the event tags. Event matches only if *all* flags are matched against the event's tags.
- **match(str)** – *optional*, match the content of an event. This is a regular expession as well.
- **order(str)** – either `asc` or `desc` - sort the results ascending or descending based on the event timestamp.

Returns an iterator over all matched results.

`theia.rdbs.`**create_store**(*db_url*, *verbose=False*)
Creates new RDBSEventStore.

Parameters

- **db_url(str)** – The database URL in SQLAlchemy form.
- **verbose(bool)** – `True` to show extended messages from the store.

Returns RDBSEventStore object.

`theia.rdbs.`**match_all**(*matchers*, *values*)
Check if *all* matchers match any of the given values.

Each matcher *must* match at least one value of the list of values.

Parameters

- **matchers(list)** – list of compiled regular expressions.

- **values (list)** – list of `str` to match

Returns `True` only if *all* of the matchers have matched at least one value of the provided list of values.

`theia.rdbs.`**`match_any`**(*matcher*, *values*)

Check if the matcher matches *any* of the supplied values.

Parameters

- **pattern)** (*matcher (regex*) – compiled regular expression.
- **values (list)** – list of `str` values to match.

Returns `True` if *any* of the `str` values matches (fullmatch) the matcher; otherwise `False`.

4.8 theia.watcher

File watcher.

Watches files and directories for changes and emits the changes as events.

class `theia.watcher.`**`DirectoryEventHandler`**(*handlers*)

Implements `watchdog.events.FileSystemEventHandler` and is used with the underlying `watchdog.observers.Observer`.

Reacts on events triggered by the watchdog Observer and passes down to the registered handlers.

The handlers are registered when creating the instance as a constructor argument. They must be specified as `dict` whose keys (`str`) are the names of the events and the entries are the event handlers themselves.

An example of creating new *`DirectoryEventHandler`*:

```
def on_file_moved(src_path, dest_path):
    print("File has moved", src_path, "->", dest_path)

event_handler = DirectoryEventHandler(handlers={
    "moved": on_file_moved
})
```

The following events are supported:

- **`moved`** **- handles the move of a file to another location. The handler** takes two arguments: the source path and the destination path. The method signature looks like this:

  ```
  def moved_handler(src_path, dest_path):
      pass
  ```

- **`created`** **- handles file creation. The handler takes one argument: the** path of the created file.

  ```
  def created_handler(file_path):
      pass
  ```

- **`modified`** **- handles file modification. The handler takes one argument:** the path of the modified file.

  ```
  def created_handler(file_path):
      pass
  ```

- **`deleted`** **- handles file deletion. The handler takes one argument: the** path of the deleted file.

```
def created_handler(file_path):
    pass
```

Parameters **handlers** (*dict*) – a `dict` of handlers for specific events.

on_created(*event*)
Called when a file or directory is created.

Parameters **event** (`DirCreatedEvent` or `FileCreatedEvent`) – Event representing file/directory creation.

on_deleted(*event*)
Called when a file or directory is deleted.

Parameters **event** (`DirDeletedEvent` or `FileDeletedEvent`) – Event representing file/directory deletion.

on_modified(*event*)
Called when a file or directory is modified.

Parameters **event** (`DirModifiedEvent` or `FileModifiedEvent`) – Event representing file/directory modification.

on_moved(*event*)
Called when a file or a directory is moved or renamed.

Parameters **event** (`DirMovedEvent` or `FileMovedEvent`) – Event representing file/directory movement.

class theia.watcher.**FileSource**(*file_path*, *callback*, *enc='UTF-8'*, *tags=None*)
Represents a source of events.

The underlying file that is being watched does not have to exist at the moment of creation of this *FileSource*.

Parameters

- **file_path** (*str*) – the path to the file to be watched.
- **callback** (*function*) – the callback handler to be executed when the file is changed. The callback is called with the difference, the path to the file and the list of tags for this source. The method signature looks like this:

  ```
  def callback(diff, path, tags):
      pass
  ```

  where:

  – `diff`, `str` is the difference from the last state of the file. Usually this is the content of the emitted event.

  – `path`, `str` is the path to the file that has changed. Usually this is the `source` property of the event.

  – `tags`, `list` is the list of tags associated with this event source.
- **enc** (*str*) – the file encoding. If not specified, `UTF-8` is assumed.
- **tags** (*list*) – list of tags associated with this source.

created()
Called when the file has actually been created. Does not trigger the callbacks.

modified()

Triggers an execution of the callbacks when the file has been modified.

Loads the difference from the source file and calls the registered callbacks.

moved(*dest_path*)

Called when the source file has been moved to another location.

Parameters **dest_path** (*str*) – the target location of the file after the move.

removed()

Called when the file has been removed. Does not triggers the callbacks.

class theia.watcher.**SourcesDaemon**(*observer*, *client*, *tags=None*)

Daemon that watches multiple sources for events.

Uses `watchdog` to monitor files and directories for changes. This defaults to using `inotify` kernel subsystem on Linux systems, `kqueue` on MacOSX and BSD-like systems and `ReadDirectoryChangesW` on Windows.

Parameters

- **observer** (*watchdog.observers.Observer*) – an instance of the `watchdog.observers.Observer` to be used.
- **client** (`theia.comm.Client`) – a client to a theia collector server.
- **tags** (*list*) – initial list of default tags that are appended to every file source watched by this daemon.

add_source(*fpath*, *enc='UTF-8'*, *tags=None*)

Add source of events to be watched by this daemon.

The path will be added as a file source and a list of tags will be associated with it. The default list of tags will be added to provided tags.

Parameters

- **fpath** (*str*) – the path of the file to be watched.
- **enc** (*str*) – the file encoding. By default `UTF-8` is assumed.
- **tags** (*list*) – list of tags to be added to the events generated by this file source.

remove_source(*fpath*)

Remove this path from the list of file event sources.

All associated watchers and handlers are removed as well.

Parameters **fpath** (*str*) – the path of the file to be removed from the watching list.

4.9 theia.cli.parser

Theia CLI main `argparse` parser.

theia.cli.parser.**get_parent_parser**(*name*, *desc=''*)

Creates the main (parent) `argparse.ArgumentParser` for Theia CLI.

Defines the main argument options such as theia server host, port, verbosity level etc.

Parameters

- **name** (*str*) – the name of the program.

- **desc** (*str*) – program description.

Returns configured `argparse.ArgumentParser`.

4.10 theia.cli.collector

Theia collector command line interface script.

`theia.cli.collector.`**`get_naive_store`**(*args*)
Creates and configures new *`theia.naivestore.NaiveEventStore`* based on the arguments passed.

Parameters **args** (*argparse.Namespace*) – arguments.

`theia.cli.collector.`**`get_parser`**(*subparsers*)
Configures the subparser for the `collect` command.

Parameters **subparser** (*argparse.ArgumentParser*) – subparser for commands.

Returns `argparse.ArgumentParser` configured for the `collect` command.

`theia.cli.collector.`**`get_rdbs_store`**(*args*)
Creates and configures new *`theia.rdbs.RDBSEventStore`* based on the arguments passed.

Parameters **args** (*argparse.Namespace*) – arguments.

`theia.cli.collector.`**`run_collector`**(*args*)
Runs the collector server.

Parameters **args** (*argparse.Namespace*) – arguments to configure the *`theia.collector.Collector`* instance.

4.11 theia.cli.watcher

Theia file watcher command line interface.

`theia.cli.watcher.`**`get_parser`**(*subparsers*)
Configures the subparser for the `watcher` command.

Parameters **subparser** (*argparse.ArgumentParser*) – subparser for commands.

Returns `argparse.ArgumentParser` configured for the `watcher` command.

`theia.cli.watcher.`**`run_watcher`**(*args*)
Runs the watcher.

Parameters **args** (*argparse.Namespace*) – the parsed arguments passed to the CLI.

4.12 theia.cli.query

Theia query command line interface.

`theia.cli.query.`**`event_printer`**(*event_format*, *time_format*, *parser*)
Builds an event printer callback with the given event format and alternative date-time format.

Parameters

- **event_format** (*str*) – the event format string.

- **`time_format`** (*`str`*) – alternative format for the event timestamp.
- **`theia.model.EventParser`** – event parser

Returns event printer that takes an event data and formats is based on the above formats.

`theia.cli.query.`**`format_event`**(*event*, *fmt*, *datefmt=None*)

Format the received event using the provided format.

Parameters

- **`event`** (`theia.model.Event`) – the event to format.
- **`fmt`** (*`str`*) – the format string. This is compatibile with `str.format()`.
- **`datefmt`** (*`str`*) – alternative date format for formatiig the event timestamp. The format must be compatible with `datetime.strftime()`

Returns the formatted event as string.

`theia.cli.query.`**`get_parser`**(*subparsers*)

Configures the subparser for the `query` command.

Parameters **`subparser`** (*`argparse.ArgumentParser`*) – subparser for commands.

Returns `argparse.ArgumentParser` configured for the `query` command.

`theia.cli.query.`**`run_query`**(*args*)

Configures and runs a *`theia.query.Query`*.

Parameters **`args`** (*`argparse.Namespace`*) – parsed command-line arguments.

4.13 theia.cli.tau

Tau is a Text User Interface frontend for Theia.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index