event-tracking Documentation

Release 0.1

edX.org

Aug 01, 2023

Contents

1 Overview **3**
1.1 Event Tracking library 3

2 User Guide **7**
2.1 Design 7
2.2 API Reference 12

3 Indices and tables **19**

Python Module Index **21**

Index **23**

Contents:

CHAPTER 1

Overview

Part of edX code.

1.1 Event Tracking library

The `event-tracking` library tracks context-aware semi-structured system events. It captures and stores events with nested data structures in order to truly take advantage of schemaless data storage systems.

Key features:

- Multiple backends - define custom backends that can be used to persist your event data.
- Nested contexts - allows data to be injected into events even without having to pass around all of said data to every location where the events are emitted.
- Django integration - provides a Django app that allows context aware events to easily be captured by multi-threaded web applications.
- MongoDB integration - support writing events out to a mongo collection.

Example:

```
from eventtracking import tracker

tracker = tracker.get_tracker()
tracker.enter_context('outer', {'user_id': 10938})
tracker.emit('navigation.request', {'url': 'http://www.edx.org/some/path/1'})

with tracker.context({'user_id': 11111, 'session_id': '2987lkjdyoioey'}):
    tracker.emit('navigation.request', {'url': 'http://www.edx.org/some/path/2'})

tracker.emit(
    'address.create',
    {
        'name': 'foo',
```

(continues on next page)

(continued from previous page)

```
        'address': {
            'postal_code': '90210',
            'country': 'United States'
        }
    }
)
```

Running the above example produces the following events:

```
{
    "name": "navigation.request",
    "timestamp": ...,
    "context": {
        "user_id": 10938
    },
    "data": {
        "url": "http://www.edx.org/some/path/1"
    }
},
{
    "name": "navigation.request",
    "timestamp": ...,
    "context": {
        "user_id": 11111,
        "session_id": "2987lkjdyoioey"
    },
    "data": {
        "url": "http://www.edx.org/some/path/2"
    }
},
{
    "name": "address.create",
    "timestamp": ...,
    "context": {
        "user_id": 10938
    },
    "data": {
        "name": "foo",
        "address": {
            "postal_code": "90210",
            "country": "United States"
        }
    }
}
```

1.1.1 Configuration

Configuration for `event-tracking` takes the form of a tree of backends. When a `Tracker` is instantiated, it creates a root `RoutingBackend` object using the top-level backends and processors that are passed to it. (Or in the case of the `DjangoTracker`, the backends and processors are constructed according to the appropriate Django settings.)

In this `RoutingBackend`, each event is first passed through the chain of processors in series, and then distributed to each backend in turn. Theoretically, these backends might be the Mongo, Segment, or logger backends, but in practice these are wrapped by another layer of `RoutingBackend`. This allows each one to have its own set of processors that are not shared with other backends, allowing independent filtering or event emit cancellation.

1.1.2 Asynchronous Routing

Considering the volume of the events being generated, we would want to avoid processing events in the main thread that could cause delays in response depending upon the operations and event processors.

`event-tracking` provides a solution for this i.e. `AsyncRoutingBackend`. It extends `RoutingBackend` but performs its operations asynchronously.

It can:

- Process event through the configured processors.
- If the event is processed successfully, pass it to the configured backends.

Handling the operations asynchronously would avoid overburdening the main thread and pass the intensive processing tasks to celery workers.

Limitations: Although backends for `RoutingBackend` can be configured at any level of `EVENT_TRACKING_BACKENDS` configuration tree, `AsyncRoutingBackend` only supports backends defined at the root level of `EVENT_TRACKING_BACKENDS` setting. It is also only possible to use it successfully from the default tracker.

An example configuration for `AsyncRoutingBackend` is provided below:

```
EVENT_TRACKING_BACKENDS = {
    'caliper': {
        'ENGINE':  'eventtracking.backends.async_routing.AsyncRoutingBackend',
        'OPTIONS': {
            'backend_name': 'caliper',
            'processors': [
                {
                    'ENGINE': 'eventtracking.processors.regex_filter.RegexFilter',
                    'OPTIONS':{
                        'filter_type': 'allowlist',
                        'regular_expressions': [
                            'edx.course.enrollment.activated',
                            'edx.course.enrollment.deactivated',
                        ]
                    }
                }
            ],
            'backends': {
                'caliper': {
                    'ENGINE': 'dummy.backend.engine',
                    'OPTIONS': {
                        ...
                    }
                }
            },
        },
    },
    'tracking_logs': {
        ...
    }
    ...
}
```

1.1.3 Roadmap

In the very near future the following features are planned:

- Dynamic event documentation and event metadata - allow event emitters to document the event types, and persist this documentation along with the events so that it can be referenced during analysis to provide context about what the event is and when it is emitted.

1.1.4 Documentation

Latest documentation (Hosted on Read the Docs)

1.1.5 License

The code in this repository is licensed under version 3 of the AGPL unless otherwise noted.

Please see `LICENSE.txt` for details.

1.1.6 How to Contribute

Contributions are very welcome.

Please read How To Contribute for details.

1.1.7 Reporting Security Issues

Please do not report security issues in public. Please email security@edx.org

1.1.8 Mailing List and IRC Channel

You can discuss this code on the edx-code Google Group or in the `edx-code` IRC channel on Freenode.

CHAPTER 2

User Guide

Note: This is a proposed design and has not yet been fully implemented in the code.

2.1 Design

2.1.1 Interface

Python

`tracker.`**`register`**(*name*, *description*, *field_descriptions*)

name A unique identification string for this type of event

description A description of the event and the conditions under which it is emitted

field_descriptions A dictionary mapping field names to a long form description

The documentation for each field is saved and used to generate navigable documentation that is shipped with the event log so that users can have some context about the various parameters in the event. Calling this method is optional, and any events emitted without first registering the event type will simply not include a reference to the event metadata.

Note: Field values can be set to any serializable object of arbitrary complexity, however, they must be documented in the documentation text for the field that will contain the object.

Any events emitted with that event type after the registration will contain a reference back to the data generated by the last call to *tracking.register()* for that event type.

Example:

```
from eventtracking import tracker

tracker.register(
    'edx.navigation.request',
    'A user visited a page',
    {
        'url': 'The url of the page visited.',
        'method': 'The HTTP method for the request, can be GET, POST, PUT, DELETE etc.
↪',
        'user_agent': 'The user agent string provided by the user's browser.'
        'parameters': 'All GET and POST parameters.  Note this excludes passwords.'
    }
)
```

`tracker.`**`emit`**(*name*, *field_values*)

name A unique identification string for an event that has already been registered.

field_values A dictionary mapping field names to the value to include in the event. Note that all values provided must be serializable.

Regardless of previous state or configuration, the data will always be logged, however, in the following conditions will cause a warning to be logged:

- the event type is unregistered
- the data contains a field that was not included in the registered event type
- the data is missing a field that was included in the registered event type
- the field_values are not serializable
- the estimated serialized event size is greater than the maximum supported

`tracker.`**`enter_context`**(*name*, *context*, *description*, *field_descriptions*)

context A dictionary of key-value pairs that will be included in every event emitted after this call. Values defined in this dictionary will override any previous calls to push_context with maps that contain the same key.

name A unique identification string for this type of context.

description A clear description of the conditions under which this context is included.

field_descriptions A dictionary mapping field names to a long form description.

Pushes a new context on to the context stack. This context will be applied to every event emitted after this call.

`tracker.`**`exit_context`**(*name*)

Removes the named context from the stack.

Javascript

`Tracker.`**`emit`**(*name*, *field_values*)

name A unique identification string for an event that has already been registered.

field_values An object mapping field names to the value to include in the event. Note that all values provided must be serializable.

See the documentation for the Python API.

Additionally, the behaviour of this function can be customized to direct events to arbitrary back-ends, and/or pre-process them before transmission to the server.

2.1.2 Event Type Metadata

The metadata for all registered event types is persisted along with a unique identifier. After registering metadata for an event type, all events emitted with that event type will contain a reference to the metadata that corresponds to that registration of the event type.

Note: The same event type may be registered multiple times with different metadata in the normal case due to revisions to the schema. This use case is supported and a new metadata record will be created for the new schema and linked to all future events of that type, while the old metadata will remain available for reference.

2.1.3 Nested Context Stack

The context stack is designed to simplify the process of including context in your events without having to have that context available at every location where the event might be emitted. It is rather cumbersome to have to pass around an HTTP request object for the sole purpose of gathering context out of it when emitting events. To aide this process you can define nested scopes which add information to the context when entered and remove information from the context when exited.

Example Scopes:

- Process
- Request
- View

Conceptually this is accomplished using a stack of dictionaries to hold all of the contexts. Contexts can be pushed on to and popped off of the stack. When an event is emitted the values for each key are included in the event metadata. Note that if multiple dictionaries on the stack contain the same key, the value from the most recently pushed context is used and the remaining values are ignored.

Example:

```
from eventtracking import tracker

tracker.enter_context('request', {'user_id': 10938})
tracker.emit('navigation.request', {'url': 'http://www.edx.org/some/path/1'})

tracker.enter_context('session', {'user_id': 11111, 'session_id': '2987lkjdyoioey'})
tracker.emit('navigation.request', {'url': 'http://www.edx.org/some/path/2'})
tracker.exit_context('session')

tracker.emit('navigation.request', {'url': 'http://www.edx.org/some/path/3'})

# The following list shows the contexts and data for the three events that are emitted
#  "context": { "user_id": 10938 }, "data": { "url": "http://www.edx.org/some/path/1"␣
↪}
#  "context": { "user_id": 11111, "session_id": "2987lkjdyoioey" }, "data": { "url":
↪"http://www.edx.org/some/path/2" }
#  "context": { "user_id": 10938 }, "data": { "url": "http://www.edx.org/some/path/3"␣
↪}
```

(continues on next page)

(continued from previous page)

2.1.4 Best Practices

- It is recommended that event types are namespaced using dot notation to avoid naming collisions, similar to DNS names. For example: edx.video.stop, mit.audio.stop
- Avoid using event type names that may cause collisions. The burden is on the analyst to decide whether your event is equivalent to another and should be grouped accordingly etc.
- Do not emit events that you don't own. This could negatively impact the analysis of the event stream. If you suspect your event is equivalent to another, say so in your documenation, and the analyst can decide whether or not to group them.

2.1.5 Sample Usage

Emitting an unregistered event:

```
tracker.emit('edx.problem.show_answer', {'problem_id': 'i4x://MITx/6.00x/problem/
↪L15:L15_Problem_2'})
```

Emitting a registered event:

```
tracker.register('edx.problem.show_answer', 'An answer was shown for a problem', {
↪'problem_id': 'A unique problem identifier'})
tracker.emit('edx.problem.show_answer', {'problem_id': 'i4x://MITx/6.00x/problem/
↪L15:L15_Problem_2'})
```

Emitting an event with context:

```
tracker.enter_context('request', {'user_id': '1234'})
try:
    tracker.emit('edx.problem.show_answer', {'problem_id': 'i4x://MITx/6.00x/problem/
↪L15:L15_Problem_2'})
finally:
    tracker.exit_context('request')
```

2.1.6 Sample Events

Show Answer:

```
{
    "name": "edx.problem.show_answer",
    "timestamp": "2013-09-12T12:55:00.12345+00:00",
    "name_id": "10ac28",
    "context_type_id": "11bd88",
    "context": {
        "course_id":"",
        "user_id": "",
        "session_id": "",
        "org_id": "",
        "origin": "client"
    }
```

(continues on next page)

(continued from previous page)

```
    "data": {
        "problem_id": "i4x://MITx/6.00x/problem/L15:L15_Problem_2"
    }
}
```

2.1.7 Sample Event Type Metadata

For the edx.problem.show_answer event type.

schema_id	name	description	timestamp	stack_trace
10ac28	edx.problem.show_answer	An answer was shown for a problem	2013-09-12T12:05:00-00:00	...
11bd88	edX context		2013-09-12T12:05:01-00:00	...

schema_field_id	schema_id	name	description
25	10ac28	problem_id	A unique problem identifier
26	11bd88	course_id	A unique course identifier
...	11bd88	...	...
40	11bd88	origin	client || server

2.1.8 Sample Event Schema

Events can be serialized into any format. Here is an example JSON serialization format that could be used to store events.

Event Schema:

```
{
    "type":"object",
    "$schema": "http://json-schema.org/draft-03/schema",
    "id": "http://edx.org/event",
    "required":true,
    "title": "Event",
    "description": "An event emitted from the edx platform.",

    "properties":{
        "name": {
            "type": "string",
            "id": "http://edx.org/event/name",
            "description": "A unique identifier for this type of event.",
            "required": true
        },
        "timestamp": {
            "type": "string",
            "id": "http://edx.org/event/timestamp",
            "description": "The UTC time the event was emitted in RFC-3339 format.",
            "required": true
        }
        "name_id": {
            "type": "string",
```

(continues on next page)

(continued from previous page)

```
            "id": "http://edx.org/event/name_id",
            "description": "A unique reference to the metadata for this event type.",
            "required": false
        },
        "context_type_id": {
            "type": "string",
            "id": "http://edx.org/event/context_type_id",
            "description": "A unique reference to the metadata for this context.",
            "required": false
        },
        "context": {
            "type": "object",
            "id": "http://edx.org/event/context",
            "description": "Context for the event that was not explicitly provided␣
↪during emission.",
            "required": false,
            "additionalProperties":true
        },
        "data": {
            "type":"object",
            "id": "http://edx.org/event/data",
            "description": "All custom fields and values provided during emission."
            "required": false,
            "additionalProperties": true
        },
    }
}
```

2.2 API Reference

2.2.1 eventtracking

A simple event tracking library

eventtracking.backends

Event tracking backend module.

eventtracking.backends.mongodb

MongoDB event tracker backend.

class eventtracking.backends.mongodb.**MongoBackend**(***kwargs*)

Bases: object

Class for a MongoDB event tracker Backend

send(*event*)

Insert the event in to the Mongo collection

eventtracking.backends.logger

Event tracker backend that saves events to a python logger.

class eventtracking.backends.logger.**DateTimeJSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.encoder.JSONEncoder

JSON encoder aware of datetime.datetime and datetime.date objects

default(*obj*)

Serialize datetime and date objects of iso format.

datatime objects are converted to UTC.

class eventtracking.backends.logger.**LoggerBackend**(**kwargs*)

Bases: object

Event tracker backend that uses a python logger.

Events are logged to the INFO level as JSON strings.

send(*event*)

Send the event to the standard python logger

eventtracking.backends.routing

Route events to processors and backends

class eventtracking.backends.routing.**RoutingBackend**(*backends=None*, *processors=None*)

Bases: object

Route events to the appropriate backends.

A routing backend has two types of components:

1) Processors - These are run sequentially, processing the output of the previous processor. If you had three processors [a, b, c], the output of the processing step would be *c(b(a(event)))*. Note that for performance reasons, the processor is able to actually mutate the event dictionary in-place. Event dictionaries may be large and highly nested, so creating multiple copies could be problematic. A processor can also choose to prevent the event from being emitted by raising *EventEmissionExit*. Doing so will prevent any subsequent processors from running and prevent the event from being sent to the backends. Any other exception raised by a processor will be logged and swallowed, subsequent processors will execute and the event will be emitted.

2) Backends - Backends are intended to not mutate the event and each receive the same event data. They are not chained like processors. Once an event has been processed by the processor chain, it is passed to each backend in the order that they were registered. Backends typically persist the event in some way, either by sending it to an external system or saving it to disk. They are called synchronously and in sequence, so a long running backend will block other backends until it is done persisting the event. Note that you can register another *RoutingBackend* as a backend of a *RoutingBackend*, allowing for arbitrary processing trees.

***backends* is a collection that supports iteration over its items using *iteritems()*. The keys are expected to be** sortable and the values are expected to expose a *send(event)* method that will be called for each event. Each backend in this collection is registered in order sorted alphanumeric ascending by key.

processors is an iterable of callables.

Raises a *ValueError* if any of the provided backends do not have a callable "send" attribute or any of the processors are not callable.

process_event(*event*)
Executes all event processors on the event in order.

event is a nested dictionary that represents the event.

Logs and swallows all *Exception* except *EventEmissionExit* which is re-raised if it is raised by a processor.

Returns the modified event.

register_backend(*name*, *backend*)
Register a new backend that will be called for each processed event.

Note that backends are called in the order that they are registered.

register_processor(*processor*)
Register a new processor.

Note that processors are called in the order that they are registered.

send(*event*)
Process the event using all registered processors and send it to all registered backends.

Logs and swallows all *Exception*.

send_to_backends(*event*)
Sends the event to all registered backends.

Logs and swallows all *Exception*.

eventtracking.backends.segment

Event tracking backend that sends events to segment.com

class eventtracking.backends.segment.**SegmentBackend**
Bases: object

Send events to segment.com

It is assumed that other code elsewhere initializes the segment.com API and makes calls to analytics.identify.

Requires all emitted events to have the following structure (at a minimum):

```
{
    'name': 'something',
    'context': {
        'user_id': 10,
    }
}
```

Additionally, the following fields can optionally be defined:

```
{
    'context': {
        'agent': "your user-agent string",
        'client_id': "your google analytics client id",
        'host': "your hostname",
        'ip': "your IP address",
        'page': "your page",
        'path': "your path",
        'referer': "your referrer",
    }
}
```

The 'page', 'path' and 'referer' are sent to Segment as "page" information. If the 'page' is absent but the 'host' and 'path' are present, these are used to create a URL value to substitute for the 'page' value.

Note that although some parts of the event are lifted out to pass explicitly into the Segment.com API, the entire event is sent as the payload to segment.com, which includes all context, data and other fields in the event.

send(*event*)
Use the segment.com python API to send the event to segment.com

eventtracking.django

eventtracking.processors

eventtracking.processors.whitelist

Filter out events whose names aren't on a pre-configured whitelist

class eventtracking.processors.whitelist.**NameWhitelistProcessor**(*whitelist=None*, ***_kwargs*)

Bases: object

Filter out events whose names aren't on a pre-configured whitelist.

whitelist is an iterable collection containing event names that should be allowed to pass.

eventtracking.processors.exceptions

Custom exceptions that are raised by this package

exception eventtracking.processors.exceptions.**EventEmissionExit**
Bases: Exception

Raising this exception indicates that no further processing of the event should occur and it should be dropped.

This should only be raised by processors.

exception eventtracking.processors.exceptions.**NoBackendEnabled**
Bases: Exception

Raise this exception when there is no backend enabled for an event.

exception eventtracking.processors.exceptions.**NoTransformerImplemented**
Bases: Exception

Raise this exception when there is no transformer implemented for an event.

eventtracking.tracker

Track application events. Supports persisting events to multiple backends.

Best Practices:

- It is recommended that event types are namespaced using dot notation to avoid naming collisions, similar to DNS names. For example: org.edx.video.stop, edu.mit.audio.stop
- Avoid using event type names that may cause collisions. The burden is on the analyst to decide whether your event is equivalent to another and should be grouped accordingly etc.
- Do not emit events that you don't own. This could negatively impact the analysis of the event stream. If you suspect your event is equivalent to another, say so in your documenation, and the analyst can decide whether or not to group them.

class eventtracking.tracker.**Tracker**(*backends=None*, *context_locator=None*, *processors=None*)

Bases: object

Track application events. Holds references to a set of backends that will be used to persist any events that are emitted.

backends
The dictionary of registered backends

context(*name*, *ctx*)
Execute the block with the given context applied. This manager ensures that the context is removed even if an exception is raised within the context.

emit(*name=None*, *data=None*)
Emit an event annotated with the UTC time when this function was called.

***name* is a unique identification string for an event that has** already been registered.

***data* is a dictionary mapping field names to the value to include in the event.** Note that all values provided must be serializable.

enter_context(*name*, *ctx*)
Enter a named context. Any events emitted after calling this method will contain all of the key-value pairs included in *ctx* unless overridden by a context that is entered after this call.

exit_context(*name*)
Exit a named context. This will remove all key-value pairs associated with this context from any events emitted after it is removed.

get_backend(*name*)
Gets the backend that was configured with *name*

located_context
The thread local context for this tracker.

processors
The list of registered processors

resolve_context()
Create a new dictionary that corresponds to the union of all of the contexts that have been entered but not exited at this point.

eventtracking.tracker.**emit**(*name=None*, *data=None*)
Calls *Tracker.emit* on the default global tracker

eventtracking.tracker.**get_tracker**(*name='default'*)
Gets a named tracker. Defaults to the default global tracker. Raises a *KeyError* if no such tracker has been registered by previously calling *register_tracker*.

eventtracking.tracker.**register_tracker**(*tracker*, *name='default'*)
Makes a tracker globally accessible. Providing no *name* parameter allows you to register the global default tracker that will be used by subsequent calls to *tracker.emit*.

eventtracking.locator

Strategies for locating contexts. Allows for arbitrarily complex caching and context differentiation strategies.

All context locators must implement a *get* method that returns an *OrderedDict*-like object.

class eventtracking.locator.**DefaultContextLocator**
Bases: object

One-to-one mapping between contexts and trackers. Every tracker will get a new context instance and it will always be returned by this locator.

get()
Get a reference to the context.

class eventtracking.locator.**ThreadLocalContextLocator**
Bases: object

Returns a different context depending on the thread that the locator was called from. Thus, contexts can be isolated from one another on thread boundaries.

Note that this makes use of *threading.local(),* which is typically monkey-patched by alternative python concurrency frameworks (like *gevent*).

Calls to *threading.local()* are delayed until first usage in order to give the third-party concurrency libraries an opportunity to monkey monkey patch it.

get()
Return a reference to a thread-specific context

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

eventtracking, 12
eventtracking.backends, 12
eventtracking.backends.logger, 13
eventtracking.backends.mongodb, 12
eventtracking.backends.routing, 13
eventtracking.backends.segment, 14
eventtracking.locator, 17
eventtracking.processors, 15
eventtracking.processors.exceptions, 15
eventtracking.processors.whitelist, 15
eventtracking.tracker, 16

Index

B

backends (*eventtracking.tracker.Tracker attribute*), 16

C

context() (*eventtracking.tracker.Tracker method*), 16

D

DateTimeJSONEncoder (*class in eventtracking.backends.logger*), 13
default() (*eventtracking.backends.logger.DateTimeJSONEncoder method*), 13
DefaultContextLocator (*class in eventtracking.locator*), 17

E

emit() (*eventtracking.tracker.Tracker method*), 16
emit() (*in module eventtracking.tracker*), 16
enter_context() (*eventtracking.tracker.Tracker method*), 16
EventEmissionExit, 15
eventtracking (*module*), 12
eventtracking.backends (*module*), 12
eventtracking.backends.logger (*module*), 13
eventtracking.backends.mongodb (*module*), 12
eventtracking.backends.routing (*module*), 13
eventtracking.backends.segment (*module*), 14
eventtracking.locator (*module*), 17
eventtracking.processors (*module*), 15
eventtracking.processors.exceptions (*module*), 15
eventtracking.processors.whitelist (*module*), 15
eventtracking.tracker (*module*), 16
exit_context() (*eventtracking.tracker.Tracker method*), 16

G

get() (*eventtracking.locator.DefaultContextLocator method*), 17
get() (*eventtracking.locator.ThreadLocalContextLocator method*), 17
get_backend() (*eventtracking.tracker.Tracker method*), 16
get_tracker() (*in module eventtracking.tracker*), 16

L

located_context (*eventtracking.tracker.Tracker attribute*), 16
LoggerBackend (*class in eventtracking.backends.logger*), 13

M

MongoBackend (*class in eventtracking.backends.mongodb*), 12

N

NameWhitelistProcessor (*class in eventtracking.processors.whitelist*), 15
NoBackendEnabled, 15
NoTransformerImplemented, 15

P

process_event() (*eventtracking.backends.routing.RoutingBackend method*), 14
processors (*eventtracking.tracker.Tracker attribute*), 16

R

register_backend() (*eventtracking.backends.routing.RoutingBackend method*), 14
register_processor() (*eventtracking.backends.routing.RoutingBackend method*), 14

register_tracker() (*in module eventtracking.tracker*), 17
resolve_context() (*eventtracking.tracker.Tracker method*), 16
RoutingBackend (*class in eventtracking.backends.routing*), 13

S

SegmentBackend (*class in eventtracking.backends.segment*), 14
send() (*eventtracking.backends.logger.LoggerBackend method*), 13
send() (*eventtracking.backends.mongodb.MongoBackend method*), 12
send() (*eventtracking.backends.routing.RoutingBackend method*), 14
send() (*eventtracking.backends.segment.SegmentBackend method*), 15
send_to_backends() (*eventtracking.backends.routing.RoutingBackend method*), 14

T

ThreadLocalContextLocator (*class in eventtracking.locator*), 17
Tracker (*class in eventtracking.tracker*), 16
Tracker.emit() (*built-in function*), 8
tracker.emit() (*built-in function*), 8
tracker.enter_context() (*built-in function*), 8
tracker.exit_context() (*built-in function*), 8
tracker.register() (*built-in function*), 7