baseplate

Jan 22, 2019

Contents

Baseplate is a framework to build services on and a library of common and well-tested code to share. Its goal is to provide all the common things a service needs with as few surprises as possible. It can be divided up into three distinct categories of components, listed below.

CHAPTER 1

The Instrumentation Framework

Baseplate provides an instrumentation framework that integrates with popular application frameworks to provide automatic diagnostics to services.

1.1 baseplate.core

The heart of the Baseplate framework is its diagnostics system. Here's an incomplete example of an application built with the framework:

```
def do_something(request):
    request.some_redis_client.ping()

def make_app(app_config):
    ... snip ...

    baseplate = Baseplate()
    baseplate.configure_metrics(metrics_client)
    baseplate.add_to_context(
        "some_redis_client", RedisContextFactory(redis_pool))

    ... snip ...
```

When a request is made which routes to the `do_something` handler, a *`ServerSpan`* is automatically created to represent the time spent processing the request in our application. If the incoming request has trace headers, the constructed server span will have the same IDs as the upstream service's child span.

When we call `request.some_redis_client.ping()` in the handler, Baseplate will create a child *`Span`* object to represent the time taken talking to redis.

The creation of the server and child spans will trigger updates on all the *`ServerSpanObserver`* and *`SpanObserver`* objects registered. Because we called `baseplate.configure_metrics` in our setup, this means we have observers that send statsd metrics so Baseplate will automatically send metrics on how long it took our application to `do_something` and how long Redis took to respond to our `ping` to statsd/Graphite without any extra code in our application.

Note: The documentation below explains how all this works under the hood. If you just want to write an application, you can skip on to *how to integrate Baseplate with your application framework* or *how to use client libraries with diagnostic instrumentation*.

1.1.1 Baseplate

At the root of each application is a single instance of `Baseplate`. This object can be integrated with various other frameworks (e.g. Thrift, Pyramid, etc.) using one of *the integrations*.

class baseplate.core.**Baseplate**

The core of the Baseplate diagnostics framework.

This class coordinates monitoring and tracing of service calls made to and from this service. See `baseplate.integration` for how to integrate it with the application framework you are using.

register(*observer*)

Register an observer.

Parameters **observer** (`baseplate.core.BaseplateObserver`) – An observer.

configure_logging()

Add request context to the logging system.

configure_metrics(*metrics_client*)

Send timing metrics to the given client.

This also adds a `baseplate.metrics.Batch` object to the `metrics` attribute on the *context object* where you can add your own application-specific metrics. The batch is automatically flushed at the end of the request.

Parameters **metrics_client** (`baseplate.metrics.Client`) – Metrics client to send request metrics to.

configure_tracing(*tracing_client*, **args*, ***kwargs*)

Collect and send span information for request tracing.

When configured, this will send tracing information automatically collected by Baseplate to the configured distributed tracing service.

Parameters **tracing_client** (`baseplate.diagnostics.tracing.TracingClient`) – Tracing client to send request traces to.

configure_error_reporting(*client*)

Send reports for unexpected exceptions to the given client.

This also adds a `raven.Client` object to the `sentry` attribute on the *context object* where you can send your own application-specific events.

Parameters **client** (`raven.Client`) – A configured raven client.

add_to_context(*name*, *context_factory*)

Add an attribute to each request's context object.

On each request, the factory will be asked to create an appropriate object to attach to the *context object*.

Parameters

- **name** (`str`) – The attribute on the context object to attach the created object to. This may also be used for metric/tracing purposes so it should be descriptive.
- **context_factory** (`baseplate.context.ContextFactory`) – A factory.

make_server_span(*context*, *name*, *trace_info=None*)
Return a server span representing the request we are handling.

In a server, a server span represents the time spent on a single incoming request. Any calls made to downstream services will be new child spans of the server span, and the server span will in turn be the child span of whatever upstream request it is part of, if any.

Parameters

- **context** – The *context object* for this request.
- **name** (*str*) – A name to identify the type of this request, e.g. a route or RPC method name.
- **trace_info** (baseplate.core.TraceInfo) – The trace context of this request as passed in from upstream. If None, a new trace context will be generated.

class baseplate.core.**TraceInfo**
Trace context for a span.

If this request was made at the behest of an upstream service, the upstream service should have passed along trace information. This class is used for collecting the trace context and passing it along to the server span.

classmethod from_upstream(*trace_id*, *parent_id*, *span_id*, *sampled*, *flags*)
Build a TraceInfo from individual headers.

Parameters

- **trace_id** (*int*) – The ID of the trace.
- **parent_id** (*int*) – The ID of the parent span.
- **span_id** (*int*) – The ID of this span within the tree.
- **sampled** (*bool*) – Boolean flag to determine request sampling.
- **flags** (*int*) – Bit flags for communicating feature flags downstream

Raises ValueError if any of the values are inappropriate.

Convenience Methods

Baseplate comes with some core monitoring observers built in and just requires you to configure them. You can enable them by calling the relevant methods on your application's *Baseplate* object.

- Logging: *configure_logging()*
- Metrics (statsd): *configure_metrics()*
- Tracing (Zipkin): *configure_tracing()*
- Error Reporting (Sentry): *configure_error_reporting()*

Additionally, Baseplate provides helpers which can be attached to the *context object* in requests. These helpers make the passing of trace information and collection of spans automatic and transparent. Because this pattern is so common, Baseplate has a special kind of observer for it which can be registered with *add_to_context()*. See the *baseplate.context* package for a list of helpers included.

1.1.2 Spans

Each time a new request comes in to be served, the time taken to handle the request is represented by a new *ServerSpan* instance. During the course of handling that request, our application might make calls to remote services or do expensive calculations, the time spent can be represented by child *Span* instances.

Spans have names and IDs and track their parent relationships. When calls are made to remote services, the information that identifies the local child span representing that service call is passed along to the remote service and becomes the server span in the remote service. This allows requests to be traced across the infrastructure.

Small bits of data, called annotations, can be attached to spans as well. This could be the URL fetched, or how many items were sent in a batch, or whatever else might be helpful.

class baseplate.core.**ServerSpan**(*trace_id*, *parent_id*, *span_id*, *sampled*, *flags*, *name*, *context*)

A server span represents a request this server is handling.

The server span is available on the *context object* during requests as the `trace` attribute.

finish(*exc_info=None*)

Record the end of the span.

Parameters **exc_info** – If the span ended because of an exception, this is the exception information. The default is `None` which indicates normal exit.

log(*name*, *payload=None*)

Add a log entry to the span.

Log entries are timestamped events recording notable moments in the lifetime of a span.

Parameters

- **name** (*str*) – The name of the log entry. This should be a stable identifier that can apply to multiple span instances.
- **payload** – Optional log entry payload. This can be arbitrary data.

make_child(*name*, *local=False*, *component_name=None*)

Return a child Span whose parent is this Span.

The child span can either be a local span representing an in-request operation or a span representing an outbound service call.

In a server, a local span represents the time spent within a local component performing an operation or set of operations. The local component is some grouping of business logic, which is then split up into operations which could each be wrapped in local spans.

Parameters

- **name** (*str*) – Name to identify the operation this span is recording.
- **local** (*bool*) – Make this span a LocalSpan if True, otherwise make this span a base Span.
- **component_name** (*str*) – Name to identify local component this span is recording in if it is a local span.

register(*observer*)

Register an observer to receive events from this span.

set_tag(*key*, *value*)

Set a tag on the span.

Tags are arbitrary key/value pairs that add context and meaning to the span, such as a hostname or query string. Observers may interpret or ignore tags as they desire.

Parameters

- **key** (*str*) – The name of the tag.
- **value** – The value of the tag, must be a string/boolean/number.

start()
Record the start of the span.

This notifies any observers that the span has started, which indicates that timers etc. should start ticking.

Spans also support the context manager protocol, for use with Python's `with` statement. When the context is entered, the span calls *start()* and when the context is exited it automatically calls *finish()*.

class baseplate.core.**Span**(*trace_id*, *parent_id*, *span_id*, *sampled*, *flags*, *name*, *context*)
A span represents a single RPC within a system.

register(*observer*)
Register an observer to receive events from this span.

start()
Record the start of the span.

This notifies any observers that the span has started, which indicates that timers etc. should start ticking.

Spans also support the context manager protocol, for use with Python's `with` statement. When the context is entered, the span calls *start()* and when the context is exited it automatically calls *finish()*.

set_tag(*key*, *value*)
Set a tag on the span.

Tags are arbitrary key/value pairs that add context and meaning to the span, such as a hostname or query string. Observers may interpret or ignore tags as they desire.

Parameters

- **key** (*str*) – The name of the tag.
- **value** – The value of the tag, must be a string/boolean/number.

log(*name*, *payload=None*)
Add a log entry to the span.

Log entries are timestamped events recording notable moments in the lifetime of a span.

Parameters

- **name** (*str*) – The name of the log entry. This should be a stable identifier that can apply to multiple span instances.
- **payload** – Optional log entry payload. This can be arbitrary data.

finish(*exc_info=None*)
Record the end of the span.

Parameters **exc_info** – If the span ended because of an exception, this is the exception information. The default is `None` which indicates normal exit.

make_child(*name*, *local=False*, *component_name=None*)
Return a child Span whose parent is this Span.

1.1.3 Observers

To actually do something with all these spans, Baseplate provides observer interfaces which receive notification of events happening in the application via calls to various methods.

The base type of observer is *BaseplateObserver* which can be registered with the root *Baseplate* instance using the *register()* method. Whenever a new server span is created in your application (i.e. a new request comes in to be served) the observer has its *on_server_span_created()* method called with the relevant details. This method can register *ServerSpanObserver* instances with the new server span to receive events as they happen.

Spans can be notified of five common events:

- *on_start()*, the span started.
- *on_set_tag()*, a tag was set on the span.
- *on_log()*, a log was entered on the span.
- *on_finish()*, the span finished.
- *on_child_span_created()*, a new child span was created.

New child spans are created in the application automatically by various client library instrumentations e.g. for a call to a remote service or database, and can also be created explicitly for local actions like expensive computations. The handler can register new *SpanObserver* instances with the new child span to receive events as they happen.

It's up to the observers to attach meaning to these events. For example, the metrics observer would start a timer `on_start()` and record the elapsed time to statsd `on_finish()`.

class baseplate.core.**BaseplateObserver**
Interface for an observer that watches Baseplate.

on_server_span_created(*context*, *server_span*)
Called when a server span is created.

Baseplate calls this when a new request begins.

Parameters

- **context** – The *context object* for this request.
- **server_span** (baseplate.core.ServerSpan) – The span representing this request.

class baseplate.core.**ServerSpanObserver**
Interface for an observer that watches the server span.

on_child_span_created(*span*)
Called when a child span is created.

SpanObserver objects call this when a new child span is created.

Parameters **span** (baseplate.core.Span) – The new child span.

on_finish(*exc_info*)
Called when the observed span is finished.

Parameters **exc_info** – If the span ended because of an exception, the exception info. Otherwise, None.

on_log(*name*, *payload*)
Called when a log entry is added to the span.

on_set_tag(*key*, *value*)
Called when a tag is set on the observed span.

on_start()
Called when the observed span is started.

class baseplate.core.**SpanObserver**
Interface for an observer that watches a span.

on_start()
Called when the observed span is started.

on_set_tag(*key*, *value*)
Called when a tag is set on the observed span.

on_log(*name*, *payload*)
Called when a log entry is added to the span.

on_finish(*exc_info*)
Called when the observed span is finished.

Parameters **exc_info** – If the span ended because of an exception, the exception info. Otherwise, None.

on_child_span_created(*span*)
Called when a child span is created.

SpanObserver objects call this when a new child span is created.

Parameters **span** (baseplate.core.Span) – The new child span.

1.1.4 Edge Request Context

The *EdgeRequestContext* provides an interface into both authentication and context information about the original request from a user. For edge services, it provides helpers to create the initial object and serialize the context information into the appropriate headers. Once this object is created and attached to the context, Baseplate will automatically forward the headers to downstream services so they can access the authentication and context data as well.

class baseplate.core.**EdgeRequestContextFactory**(*secrets*)
Factory for creating *EdgeRequestContext* objects.

Every application should set one of these up. Edge services that talk directly with clients should use *new()* directly. For internal services, pass the object off to Baseplate's framework integration (Thrift/Pyramid) for automatic use.

Parameters **secrets** (baseplate.secrets.SecretsStore) – A configured secrets store.

new(*authentication_token=None*, *loid_id=None*, *loid_created_ms=None*, *session_id=None*)
Return a new EdgeRequestContext object made from scratch.

Services at the edge that communicate directly with clients should use this to pass on the information they get to downstream services. They can then use this information to check authentication, run experiments, etc.

To use this, create and attach the context early in your request flow:

```
auth_cookie = request.cookies["authentication"]
token = request.authentication_service.authenticate_cookie(cookie)
loid = parse_loid(request.cookies["loid"])
session = parse_session(request.cookies["session"])

edge_context = self.edgecontext_factory.new(
    authentication_token=token,
    loid_id=loid.id,
    loid_created_ms=loid.created,
    session_id=session.id,
)
edge_context.attach_context(request)
```

Parameters

- **authentication_token** – (Optional) A raw authentication token as returned by the authentication service.
- **loid_id** (*str*) – (Optional) ID for the current LoID in fullname format.
- **loid_created_ms** (*int*) – (Optional) Epoch milliseconds when the current LoID cookie was created.
- **session_id** (*str*) – (Optional) ID for the current session cookie.

from_upstream(*edge_header*)
Create and return an EdgeRequestContext from an upstream header.

This is generally used internally to Baseplate by framework integrations that automatically pick up context from inbound requests.

Parameters **edge_header** – Raw payload of Edge-Request header from upstream service.

class baseplate.core.**EdgeRequestContext**(*authn_token_validator*, *header*)
Contextual information about the initial request to an edge service

Construct this using an *EdgeRequestContextFactory*.

attach_context(*context*)
Attach this to the provided *context object*.

Parameters **context** – request context to attach this to

event_fields()
Return fields to be added to events.

user
User object for the current context

oauth_client
OAuthClient object for the current context

session
Session object for the current context

service
Service object for the current context

class baseplate.core.**User**
Wrapper for the user values in AuthenticationToken and the LoId cookie.

id
Authenticated account_id for the current User.

Type account_id string or None if context authentication is invalid

Raises *NoAuthenticationError* if there was no authentication token, it was invalid, or the subject is not an account.

is_logged_in
Does the User have a valid, authenticated id?

roles
Authenticated roles for the current User.

Type set(string)

Raises *NoAuthenticationError* if there was no authentication token or it was invalid

has_role(*role*)
Does the authenticated user have the specified role?

Parameters **client_types** (*str*) – Case-insensitive sequence role name to check.

Type bool

Raises *NoAuthenticationError* if there was no authentication token defined for the current context

event_fields()
Return fields to be added to events.

class baseplate.core.**OAuthClient**
Wrapper for the OAuth2 client values in AuthenticationToken.

id
Authenticated id for the current client

Type string or None if context authentication is invalid

Raises *NoAuthenticationError* if there was no authentication token defined for the current context

is_type(**client_types*)
Is the authenticated client type one of the given types?

When checking the type of the current OauthClient, you should check that the type "is" one of the allowed types rather than checking that it "is not" a disallowed type.

For example:

```
if oauth_client.is_type("third_party"):
    ...
```

not:

```
if not oauth_client.is_type("first_party"):
    ...
```

Parameters **client_types** (*str*) – Case-insensitive sequence of client type names that you want to check.

Type bool

Raises *NoAuthenticationError* if there was no authentication token defined for the current context

event_fields()
Return fields to be added to events.

class baseplate.core.**Session**(*id*)

class baseplate.core.**AuthenticationToken**
Information about the authenticated user.

EdgeRequestContext provides high-level helpers for extracting data from authentication tokens. Use those instead of direct access through this class.

subject
The raw *subject* that is authenticated.

exception baseplate.core.**NoAuthenticationError**
Raised when trying to use an invalid or missing authentication token.

1.2 baseplate.context

Helpers that integrate common client libraries with baseplate's diagnostics.

This package contains modules which integrate various client libraries with Baseplate's instrumentation. When using these client library integrations, trace information is passed on and metrics are collected automatically.

To use these helpers, use the *add_to_context()* method on your application's *Baseplate* object:

```
client = SomeClient("server, server, server")
baseplate.add_to_context("my_client", SomeContextFactory(client))
```

and then a context-aware version of the client will show up on the *context object* during requests:

```
def my_handler(self, context):
    context.my_client.make_some_remote_call()
```

1.2.1 Instrumented Client Libraries

baseplate.context.cassandra

Configuration Parsing

baseplate.context.cassandra.**cluster_from_config**(*app_config*, *prefix=u'cassandra.'*, ***kwargs*)
Make a Cluster from a configuration dictionary.

The keys useful to *cluster_from_config()* should be prefixed, e.g. `cassandra.contact_points` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `Cluster` constructor. Any keyword arguments given to this function will be passed through to the `Cluster` constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- `contact_points` (required): comma delimited list of contact points to try connecting for cluster discovery
- `port`: The server-side port to open connections to.

Classes

class baseplate.context.cassandra.**CassandraContextFactory**(*session*)
Cassandra session context factory.

This factory will attach a proxy object which acts like a `cassandra.cluster.Session` to an attribute on the *context object*. The `execute`, `execute_async` or `prepare` methods will automatically record diagnostic information.

Parameters **session** (*cassandra.cluster.Session*) – A configured session object.

class baseplate.context.cassandra.**CQLMapperContextFactory**(*session*)
CQLMapper ORM connection context factory

This factory will attach a new CQLMapper `cqlmapper.connection.Connection` to an attribute on the *context object*. This Connection object will use the same proxy object that CassandraContextFactory attaches to a context to run queries so the *execute* command will automatically record diagnostic information.

Parameters **session** (*cassandra.cluster.Session*) – A configured session object.

baseplate.context.hvac

Integration with HVAC, a Vault Python client, for advanced Vault features.

See HVAC's README for documentation on the methods available from its client.

Note: The *SecretsStore* handles the most common use case of Vault in a Baseplate application: secure retrieval of secret tokens. This client is only necessary when taking advantage of more advanced features of Vault such as the Transit backend or Cubbyholes. If these don't sound familiar, check out the secrets store before digging in here.

Configuration Parsing

baseplate.context.hvac.**hvac_factory_from_config**(*app_config*, *secrets_store*, *prefix=u'vault.'*)
Make an HVAC client factory from a configuration dictionary.

The keys useful to *hvac_factory_from_config()* should be prefixed, e.g. `vault.timeout`. The `prefix` argument specifies the prefix used to filter keys.

Supported keys:

- `timeout`: How long to wait for calls to Vault.

Parameters

- **app_config** (*dict*) – The raw application configuration.
- **secrets_store** (baseplate.secrets.SecretsStore) – A configured secrets store from which we can get a Vault authentication token.
- **prefix** (*str*) – The prefix for configuration keys.

Classes

class baseplate.context.hvac.**HvacContextFactory**(*secrets_store*, *timeout*)
HVAC client context factory.

This factory will attach a proxy object which acts like an `hvac.Client` to an attribute on the *context object*. All methods that talk to Vault will be automatically instrumented for tracing and diagnostic metrics.

Parameters

- **secrets_store** (baseplate.secrets.SecretsStore) – Configured secrets store from which we can get a Vault authentication token.
- **timeout** (*datetime.timedelta*) – How long to wait for calls to Vault.

baseplate.context.kombu

Configuration Parsing

baseplate.context.kombu.**connection_from_config**(*app_config*, *prefix*, ***kwargs*)

Make a Connection from a configuration dictionary.

The keys useful to *connection_from_config()* should be prefixed, e.g. `amqp.hostname` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `Connection` constructor. Any keyword arguments given to this function will be passed through to the `Connection` constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- `hostname`
- `virtual_host`

baseplate.context.kombu.**exchange_from_config**(*app_config*, *prefix*, ***kwargs*)

Make an Exchange from a configuration dictionary.

The keys useful to *exchange_from_config()* should be prefixed, e.g. `amqp.exchange_name` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `Exchange` constructor. Any keyword arguments given to this function will be passed through to the `Exchange` constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- `exchange_name`
- `exchange_type`

Classes

class baseplate.context.kombu.**KombuProducerContextFactory**(*connection*, *exchange*, *max_connections=None*)

KombuProducer context factory.

This factory will attach a proxy object which acts like a `kombu.Producer` to an attribute on the *context object*. The *publish()* method will automatically record diagnostic information.

Parameters

- **connection** (*kombu.connection.Connection*) – A configured connection object.
- **exchange** (*kombu.Exchange*) – A configured exchange object
- **max_connections** (*int*) – The maximum number of connections.

class baseplate.context.kombu.**KombuProducer**(*name*, *span*, *connection*, *exchange*, *producers*)

publish(*body*, *routing_key=None*, ***kwargs*)

Publish a message to the routing_key.

Parameters

- **body** (*str*) – The message body.
- **routing_key** (*str*) – The routing key to publish to.

See Kombu Documentation for other arguments.

baseplate.context.memcache

Configuration Parsing

baseplate.context.memcache.**pool_from_config**(*app_config*, *prefix=u'memcache.'*, *serializer=None*, *deserializer=None*)

Make a PooledClient from a configuration dictionary.

The keys useful to *pool_from_config()* should be prefixed, e.g. `memcache.endpoint`, `memcache.max_pool_size`, etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `PooledClient` constructor.

Supported keys:

- **`endpoint` (required): a string representing a host and port to connect** to memcached service, e.g. `localhost:11211` or `127.0.0.1:11211`.
- **`max_pool_size`: an integer for the maximum pool size to use, by default** this is `2147483648`.
- **`connect_timeout`: a float representing seconds to wait for a connection to** memcached server. Defaults to the underlying socket default timeout.
- **`timeout`: a float representing seconds to wait for calls on the** socket connected to memcache. Defaults to the underlying socket default timeout.

Parameters

- **`app_config`** (*dict*) – the config dictionary
- **`prefix`** (*str*) – prefix for config keys
- **`serializer`** (*callable*) – function to serialize values to strings suitable for being stored in memcached. An example is *make_dump_and_compress_fn()*.
- **`deserializer`** (*callable*) – function to convert strings returned from memcached to arbitrary objects, must be compatible with `serializer`. An example is *decompress_and_load()*.

Returns `pymemcache.client.base.PooledClient`

Classes

class baseplate.context.memcache.**MemcacheContextFactory**(*pooled_client*)

Memcache client context factory.

This factory will attach a *MonitoredMemcacheConnection* to an attribute on the *context object*. When memcache commands are executed via this connection object, they will use connections from the provided `PooledClient` and automatically record diagnostic information.

Parameters **`pooled_client`** (*pymemcache.client.base.PooledClient*) – A pooled client.

Returns *MonitoredMemcacheConnection*

class baseplate.context.memcache.**MonitoredMemcacheConnection**(*context_name*, *server_span*, *pooled_client*)

Memcache connection that collects diagnostic information.

This connection acts like a `PooledClient` except that operations are wrapped with diagnostic collection. Some methods may not yet be wrapped with monitoring. Please request assistance if any needed methods are not being monitored.

Serialization/deserialization helpers

baseplate.context.memcache.lib.**decompress_and_load**(*key*, *serialized*, *flags*)

Deserialize data.

This should be paired with *make_dump_and_compress_fn()*.

Parameters

- **key** (*str*) – the memcached key.
- **serialized** (*str*) – the serialized object returned from memcached.
- **flags** (*int*) – value stored and returned from memcached for the client to use to indicate how the value was serialized.

Returns The deserialized value.

baseplate.context.memcache.lib.**make_dump_and_compress_fn**(*min_compress_length=0*, *compress_level=1*)

Make a serializer.

This should be paired with *decompress_and_load()*.

The resulting method is a chain of `json.loads()` and `zlib` compression. Values that are not JSON serializable will result in a `TypeError`.

Parameters

- **min_compress_length** (*int*) – the minimum serialized string length to enable zlib compression. 0 disables compression.
- **compress_level** (*int*) – zlib compression level. 0 disables compression and 9 is the maximum value.

Returns The serializer.

baseplate.context.memcache.lib.**decompress_and_unpickle**(*key*, *serialized*, *flags*)

Deserialize data stored by `pylibmc`.

> **Warning:** This should only be used when sharing caches with applications using `pylibmc` (like r2). New applications should use the safer and future proofed *decompress_and_load()*.

Parameters

- **key** (*str*) – the memcached key.
- **serialized** (*str*) – the serialized object returned from memcached.
- **flags** (*int*) – value stored and returned from memcached for the client to use to indicate how the value was serialized.

Returns str value the deserialized value.

baseplate.context.memcache.lib.**make_pickle_and_compress_fn**(*min_compress_length=0*, *compress_level=1*)

Make a serializer compatible with `pylibmc` readers.

The resulting method is a chain of `pickle.dumps()` and `zlib` compression. This should be paired with *decompress_and_unpickle()*.

Warning: This should only be used when sharing caches with applications using `pylibmc` (like r2). New applications should use the safer and future proofed *make_dump_and_compress_fn()*.

Parameters

- **min_compress_length** (*int*) – the minimum serialized string length to enable zlib compression. 0 disables compression.
- **compress_level** (*int*) – zlib compression level. 0 disables compression and 9 is the maximum value.

Returns func memcache_serializer the serializer method.

baseplate.context.redis

Configuration Parsing

Classes

baseplate.context.sqlalchemy

Configuration Parsing

sqlalchemy.**engine_from_config**(*configuration*, *prefix='sqlalchemy.'*, ***kwargs*)

Make an engine from a configuration dictionary.

The keys useful to *engine_from_config()* should be prefixed, e.g. `sqlalchemy.url` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on `create_engine()`. Any keyword arguments given to this function will be passed through. Keyword arguments take precedence over the configuration file.

Classes

class baseplate.context.sqlalchemy.**SQLAlchemyEngineContextFactory**(*engine*)

SQLAlchemy core engine context factory.

This factory will attach a SQLAlchemy `sqlalchemy.engine.Engine` to an attribute on the *context object*. All cursor (query) execution will automatically record diagnostic information.

Additionally, the trace and span ID will be added as a comment to the text of the SQL statement. This is to aid correlation of queries with requests.

See also:

The engine is the low-level SQLAlchemy API. If you want to use the ORM, consider using *SQLAlchemySessionContextFactory* instead.

Parameters **engine** (*sqlalchemy.engine.Engine*) – A configured SQLAlchemy engine.

class baseplate.context.sqlalchemy.**SQLAlchemySessionContextFactory**(*engine*)
SQLAlchemy ORM session context factory.

This factory will attach a new SQLAlchemy `sqlalchemy.orm.session.Session` to an attribute on the *context object*. All cursor (query) execution will automatically record diagnostic information.

The session will be automatically closed, but not committed or rolled back, at the end of each request.

See also:

The session is part of the high-level SQLAlchemy ORM API. If you want to do raw queries, consider using *SQLAlchemyEngineContextFactory* instead.

Parameters **engine** (*sqlalchemy.engine.Engine*) – A configured SQLAlchemy engine.

baseplate.context.thrift

class baseplate.context.thrift.**ThriftContextFactory**(*pool*, *client_cls*)
Thrift client pool context factory.

This factory will attach a proxy object with the same interface as your thrift client to an attribute on the *context object*. When a thrift method is called on this proxy object, it will check out a connection from the connection pool and execute the RPC, automatically recording diagnostic information.

Parameters

- **pool** (`baseplate.thrift_pool.ThriftConnectionPool`) – The connection pool.
- **client_cls** – The class object of a Thrift-generated client class, e.g. `YourService.Client`.

The proxy object has a `retrying` method which takes the same parameters as *RetryPolicy.new* and acts as a context manager. The context manager returns another proxy object where Thrift service method calls will be automatically retried with the specified retry policy when transient errors occur:

```
with context.my_service.retrying(attempts=3) as svc:
    svc.some_method()
```

1.2.2 DIY: The Factory

If a library you want isn't supported here, it can be added to your own application by implementing *ContextFactory*.

class baseplate.context.**ContextFactory**
An interface for adding stuff to the context object.

Objects implementing this interface can be passed to *add_to_context()*. The return value of *make_object_for_context()* will be added to the *context object* with the name specified in `add_to_context`.

make_object_for_context(*name*, *server_span*)
Return an object that can be added to the context object.

1.3 baseplate.integration

Helpers for integration with various application frameworks.

This package contains modules which integrate Baseplate with common application frameworks.

See one of the submodules below for your framework of choice.

1.3.1 Thrift

Thrift integration for Baseplate.

This module provides an implementation of `TProcessorEventHandler` which integrates Baseplate's facilities into the Thrift request lifecycle.

An abbreviated example of it in use:

```
def make_processor(app_config):
    baseplate = Baseplate()

    handler = MyHandler()
    processor = my_thrift.MyService.ContextProcessor(handler)

    event_handler = BaseplateProcessorEventHandler(logger, baseplate)
    processor.setEventHandler(event_handler)

    return processor
```

class baseplate.integration.thrift.**BaseplateProcessorEventHandler**(*logger*, *baseplate*, *edge_context_factory=None*)

Processor event handler for Baseplate.

Parameters

- **logger** (*logging.Logger*) – The logger to use for error and debug logging.
- **baseplate** (baseplate.core.Baseplate) – The baseplate instance for your application.
- **edge_context_factory** (baseplate.core.EdgeRequestContextFactory) – A configured factory for handling edge request context.

1.3.2 Pyramid

Pyramid integration for Baseplate.

This module provides a configuration extension for Pyramid which integrates Baseplate's facilities into the Pyramid WSGI request lifecycle.

An abbreviated example of it in use:

```
def make_app(app_config):
    configurator = Configurator()

    baseplate = Baseplate()
    baseplate_config = BaseplateConfigurator(
        baseplate,
```

(continues on next page)

(continued from previous page)

```
        trust_trace_headers=True,
    )
    configurator.include(baseplate_config.includeme)

    return configurator.make_wsgi_app()
```

Warning: Because of how Baseplate instruments Pyramid, you should not make an exception view prevent Baseplate from seeing the unhandled error and reporting it appropriately.

class baseplate.integration.pyramid.**BaseplateConfigurator**(*baseplate*, *trust_trace_headers=False*, *edge_context_factory=None*)

Config extension to integrate Baseplate into Pyramid.

Parameters

- **baseplate** (baseplate.core.Baseplate) – The Baseplate instance for your application.
- **trust_trace_headers** (*bool*) – Should this application trust trace headers from the client? If `True`, trace headers in inbound requests will be used for the server span. If `False`, new random trace IDs will be generated for each request.
- **edge_context_factory** (baseplate.core.EdgeRequestContextFactory) – A configured factory for handling edge request context.

Warning: Do not set `trust_trace_headers` to `True` unless you are sure your application is only accessible by trusted sources (usually backend-only services).

Events

Within its Pyramid integration, Baseplate will emit events at various stages of the request lifecycle that services can hook into.

class baseplate.integration.pyramid.**ServerSpanInitialized**(*request*)

Event that Baseplate fires after creating the ServerSpan for a Request.

This event will be emitted before the Request is passed along to it's handler. Baseplate initializes the ServerSpan in response to a `pyramid.events.ContextFound` event emitted by Pyramid so while we can guarantee what Baseplate has done when this event is emitted, we cannot guarantee that any other subscribers to `pyramid.events.ContextFound` have been called or not.

1.4 baseplate.diagnostics

1.4.1 Observers

Observers watch Baseplate for events that happen during requests, such as requests starting and ending and service calls being made. Observers can also add attributes to the *context object* for your application to use during the request. Under the hood, the context factories (*baseplate.context*) are implemented as observers. All of the following observers can be configured with *Convenience Methods* on your application's *Baseplate* object.

class baseplate.diagnostics.logging.**LoggingBaseplateObserver**
Logging observer.

This observer adds request context to the thread-local state so that the log formatters can give more informative logs. Currently, this just sets the thread name to the current request's trace ID.

class baseplate.diagnostics.metrics.**MetricsBaseplateObserver**(*client*)
Metrics collecting observer.

This observer reports metrics to statsd. It does two important things:

- it tracks the time taken in serving each request.
- it batches all metrics generated during a request into as few packets as possible.

The batch is accessible to your application during requests as the `metrics` attribute on the *context object*.

Parameters **client** (`baseplate.metrics.Client`) – The client where metrics will be sent.

class baseplate.diagnostics.sentry.**SentryBaseplateObserver**(*raven*)
Error reporting observer.

This observer reports unexpected exceptions to Sentry.

The raven client is accessible to your application during requests as the `sentry` attribute on the *context object*.

Parameters **client** (*raven.Client*) – A configured raven client.

class baseplate.diagnostics.tracing.**TraceBaseplateObserver**(*tracing_client*)
Distributed tracing observer.

This observer handles Zipkin-compatible distributed tracing instrumentation for both inbound and outbound requests. Baseplate span-specific tracing observers (TraceSpanObserver and TraceServerSpanObserver) are registered for tracking, serializing, and recording span data.

Parameters **client** (*baseplate.diagnostics.tracing.TracingClient*) – The client where metrics will be sent.

CHAPTER 2

The Library

Baseplate also provides a collection of "extra batteries". These independent modules provide commonly needed functionality to applications. They can be used separately from the rest of Baseplate.

2.1 baseplate

baseplate.**metrics_client_from_config**(*raw_config*)
Configure and return a metrics client.

This expects two configuration options:

metrics.namespace The root key to prefix all metrics in this application with.

metrics.endpoint A `host:port` pair, e.g. `localhost:2014`. If an empty string, a client that discards all metrics will be returned.

Parameters **raw_config** (*dict*) – The application configuration which should have settings for the metrics client.

Returns A configured client.

Return type *baseplate.metrics.Client*

baseplate.**tracing_client_from_config**(*raw_config*, *log_if_unconfigured=True*)
Configure and return a tracing client.

This expects one configuration option and can take many optional ones:

tracing.service_name The name for the service this observer is registered to.

tracing.endpoint (optional) (Deprecated in favor of the sidecar model.) Destination to record span data.

tracing.queue_name (optional) Name of POSIX queue where spans are recorded

tracing.max_span_queue_size (optional) Span processing queue limit.

tracing.num_span_workers (optional) Number of worker threads for span processing.

tracing.span_batch_interval (optional) Wait time for span processing in seconds.

tracing.num_conns (optional) Pool size for remote recorder connection pool.

tracing.sample_rate (optional) Percentage of unsampled requests to record traces for (e.g. "37%")

Parameters

- **raw_config** (*dict*) – The application configuration which should have settings for the tracing client.
- **log_if_unconfigured** (*bool*) – When the client is not configured, should trace spans be logged or discarded silently?

Returns A configured client.

Return type baseplate.diagnostics.tracing.TracingClient

baseplate.**error_reporter_from_config**(*raw_config*, *module_name*)

Configure and return a error reporter.

This expects one configuration option and can take many optional ones:

sentry.dsn The DSN provided by Sentry. If blank, the reporter will discard events.

sentry.site (optional) An arbitrary string to identify this client installation.

sentry.environment (optional) The environment your application is running in.

sentry.exclude_paths (optional) Comma-delimited list of module prefixes to ignore when discovering where an error came from.

sentry.include_paths (optional) Comma-delimited list of paths to include for consideration when drilling down to an exception.

sentry.ignore_exceptions (optional) Comma-delimited list of fully qualified names of exception classes (potentially with * globs) to not report.

sentry.sample_rate (optional) Percentage of errors to report. (e.g. "37%")

sentry.processors (optional) Comma-delimited list of fully qualified names of processor classes to apply to events before sending to Sentry.

Example usage:

```
error_reporter_from_config(app_config, __name__)
```

Parameters

- **raw_config** (*dict*) – The application configuration which should have settings for the error reporter.
- **module_name** (*str*) – __name__ of the root module of the application.

Return type raven.Client

2.2 baseplate.config

Configuration parsing and validation.

This module provides parse_config which turns a dictionary of stringy keys and values into a structured and typed configuration object.

For example, an INI file like the following:

```
[app:main]
simple = true
cards = clubs, spades, diamonds
nested.once = 1
nested.really.deep = 3 seconds
some_file = /var/lib/whatever.txt
sample_rate = 37.1%
interval = 30 seconds
```

Might be parsed like the following. Note: when running under the baseplate server, The `config_parser.items(...)` step is taken care of for you and `raw_config` is passed as the only argument to your factory function.

```
>>> raw_config = dict(config_parser.items("app:main"))

>>> CARDS = config.OneOf(clubs=1, spades=2, diamonds=3, hearts=4)
>>> cfg = config.parse_config(raw_config, {
...     "simple": config.Boolean,
...     "cards": config.TupleOf(CARDS),
...     "nested": {
...         "once": config.Integer,
...
...         "really": {
...             "deep": config.Timespan,
...         },
...     },
...     "some_file": config.File(mode="r"),
...     "optional": config.Optional(config.Integer, default=9001),
...     "sample_rate": config.Percent,
...     "interval": config.Fallback(config.Timespan, config.Integer),
... })

>>> print(cfg.simple)
True

>>> print(cfg.cards)
[1, 2, 3]

>>> print(cfg.nested.really.deep)
0:00:03

>>> cfg.some_file.read()
'cool'
>>> cfg.some_file.close()

>>> cfg.sample_rate
0.371

>>> print(cfg.interval)
0:00:30
```

2.2.1 Parser

baseplate.config.**parse_config**(*config*, *spec*)

Parse options against a spec and return a structured representation.

Parameters

- **config** (*dict*) – The raw stringy configuration dictionary.
- **spec** (*dict*) – A specification of what the config should look like.

Raises *ConfigurationError* The configuration violated the spec.

Returns A structured configuration object.

2.2.2 Value Types

Each option can have a type specified. Some types compose with other types to make complicated expressions.

baseplate.config.**String**(*text*)
A raw string.

baseplate.config.**Float**(*text*)
A floating-point number.

baseplate.config.**Integer**(*text=None*, *base=10*)
An integer.

To prevent mistakes, this will raise an error if the user attempts to configure a non-whole number.

Parameters **base** (*int*) – (Optional) If specified, the base of the integer to parse.

baseplate.config.**Boolean**(*text*)
True or False, case insensitive.

baseplate.config.**Endpoint**(*text*)
A remote endpoint to connect to.

Returns an *EndpointConfiguration*.

If the endpoint is a hostname:port pair, the `family` will be `socket.AF_INET` and `address` will be a two-tuple of host and port, as expected by `socket`.

If the endpoint contains a slash (`/`), it will be interpreted as a path to a UNIX domain socket. The `family` will be `socket.AF_UNIX` and `address` will be the path as a string.

baseplate.config.**Timespan**(*text*)
A span of time.

This takes a string of the form "1 second" or "3 days" and returns a `datetime.timedelta` representing that span of time.

Units supported are: milliseconds, seconds, minutes, hours, days.

baseplate.config.**Base64**(*text*)
A base64 encoded block of data.

This is useful for arbitrary binary blobs.

baseplate.config.**File**(*mode=u'r'*)
A path to a file.

This takes a path to a file and returns an open file object, like returned by `open()`.

Parameters **mode** (*str*) – an optional string that specifies the mode in which the file is opened.

baseplate.config.**Percent**(*text*)
A percentage.

This takes a string of the form "37.2%" or "44%" and returns a float in the range [0.0, 1.0].

baseplate.config.**UnixUser**(*text*)
A Unix user name.

The parsed value will be the integer user ID.

baseplate.config.**UnixGroup**(*text*)
A Unix group name.

The parsed value will be the integer group ID.

baseplate.config.**OneOf**(***options*)
One of several choices.

For each `option`, the name is what should be in the configuration file and the value is what it is mapped to.

For example:

```
OneOf(hearts="H", spades="S")
```

would parse:

```
"hearts"
```

into:

```
"H"
```

baseplate.config.**TupleOf**(*T*)
A comma-delimited list of type T.

At least one value must be provided. If you want an empty list to be a valid choice, wrap with *`Optional()`*.

If you need something custom or fancy for your application, just use a callable which takes a string and returns the parsed value or raises `ValueError`.

2.2.3 Combining Types

These options are used in combination with other types to form more complex configurations.

baseplate.config.**Optional**(*T*, *default=None*)
An option of type T, or `default` if not configured.

baseplate.config.**Fallback**(*T1*, *T2*)
An option of type T1, or if that fails to parse, of type T2.

This is useful for backwards-compatible configuration changes.

baseplate.config.**DictOf**(*spec*)
A group of options of a given type.

This is useful for providing data to the application without the application having to know ahead of time all of the possible keys.

```
[app:main]
population.cn = 1383890000
population.in = 1317610000
population.us = 325165000
population.id = 263447000
population.br = 207645000
```

```
>>> cfg = config.parse_config(raw_config, {
...     "population": config.DictOf(config.Integer),
... })

>>> len(cfg.population)
5

>>> cfg.population["br"]
207645000
```

It can also be combined with other configuration specs or parsers to parse more complicated structures:

```
[app:main]
countries.cn.population = 1383890000
countries.cn.capital = Beijing
countries.in.population = 1317610000
countries.in.capital = New Delhi
countries.us.population = 325165000
countries.us.capital = Washington D.C.
countries.id.population = 263447000
countries.id.capital = Jakarta
countries.br.population = 207645000
countries.br.capital = Brasília
```

```
>>> cfg = config.parse_config(raw_config, {
...     "countries": config.DictOf({
...         "population": config.Integer,
...         "capital": config.String,
...     }),
... })

>>> len(cfg.countries)
5

>>> cfg.countries["cn"].capital
'Beijing'

>>> cfg.countries["id"].population
263447000
```

2.2.4 Data Types

class baseplate.config.**EndpointConfiguration**

A description of a remote endpoint.

This is a 2-tuple of (`family` and `address`).

family One of `socket.AF_INET` or `socket.AF_UNIX`.

address An address appropriate for the `family`.

See also:

baseplate.config.Endpoint()

2.2.5 Exceptions

exception baseplate.config.**ConfigurationError**(*key*, *error*)
Raised when the configuration violates the spec.

2.3 baseplate.crypto

Utilities for common cryptographic operations.

```
message = "Hello, world!"

secret = secrets.get_versioned("some_signing_key")
signature = make_signature(
    secret, message, max_age=datetime.timedelta(days=1))

try:
    validate_signature(secret, message, signature)
except SignatureError:
    print("Oh no, it was invalid!")
else:
    print("Message was valid!")
```

```
Message was valid!
```

2.3.1 Message Signing

baseplate.crypto.**make_signature**(*secret*, *message*, *max_age*)
Return a signature for the given message.

To ensure that key rotation works automatically, always fetch the secret token from the secret store immediately before use and do not cache / save the token anywhere. The `current` version of the secret will be used to sign the token.

Parameters

- **secret** (baseplate.secrets.VersionedSecret) – The secret signing key from the secret store.
- **message** (*str*) – The message to sign.
- **max_age** (*datetime.timedelta*) – The amount of time in the future the signature will be valid for.

Returns An encoded signature.

baseplate.crypto.**validate_signature**(*secret*, *message*, *signature*)
Validate and assert a message's signature is correct.

If the signature is valid, the function will return normally with a *SignatureInfo* with some details about the signature. Otherwise, an exception will be raised.

To ensure that key rotation works automatically, always fetch the secret token from the secret store immediately before use and do not cache / save the token anywhere. All active versions of the secret will be checked when validating the signature.

Parameters

- **secret** (baseplate.secrets.VersionedSecret) – The secret signing key from the secret store.
- **message** (*str*) – The message payload to validate.
- **signature** (*str*) – The signature supplied with the message.

Raises *UnreadableSignatureError* The signature is corrupt.

Raises *IncorrectSignatureError* The digest is incorrect.

Raises *ExpiredSignatureError* The signature expired.

Return type *SignatureInfo*

class baseplate.crypto.**SignatureInfo**
Information about a valid signature.

Variables

- **version** (*int*) – The version of the packed signature format.
- **expiration** (*int*) – The time, in seconds since the UNIX epoch, at which the signature will expire.

Exceptions

exception baseplate.crypto.**SignatureError**
Base class for all message signing related errors.

exception baseplate.crypto.**UnreadableSignatureError**
Raised when the signature is corrupt or wrongly formatted.

exception baseplate.crypto.**IncorrectSignatureError**
Raised when the signature is readable but does not match the message.

exception baseplate.crypto.**ExpiredSignatureError**(*expiration*)
Raised when the signature is valid but has expired.

The `expiration` attribute is the time (as seconds since the UNIX epoch) at which the signature expired.

2.3.2 Utilities

baseplate.crypto.**constant_time_compare**()
compare_digest(a, b) -> bool

Return 'a == b'. This function uses an approach designed to prevent timing analysis, making it appropriate for cryptography. a and b must both be of the same type: either str (ASCII only), or any type that supports the buffer protocol (e.g. bytes).

Note: If a and b are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of a and b–but not their values.

2.4 baseplate.events

2.4.1 Building Events

Thrift Schema v2 Events

For modern Thrift-based events: import the event schemas into your project, instantiate and fill out an event object, and pass it into the queue:

```
import time
import uuid

from baseplate.events import EventQueue, serialize_v2_event

from event_schemas.event.ttypes import Event


def make_wsgi_app(app_config):
    ...

    queue = EventQueue("v2", event_serializer=serialize_v2_event)
    baseplate.add_to_context("events_v2", queue)

    ...


def my_handler(request):
    event = Event(
        source="baseplate",
        action="test",
        noun="baseplate",
        client_timestamp=time.time() * 1000,
        uuid=str(uuid.uuid4()),
    )
    request.events_v2.put(ev2)
```

Legacy schemaless events

For legacy schemaless events, you can use these helper objects to build payloads:

class baseplate.events.**FieldKind**
Field kinds.

NORMAL = None
For fields normal fields with no hashing/indexing requirements.

OBFUSCATED = u'obfuscated_data'
For fields containing sensitive information like IP addresses that must be treated with care.

HIGH_CARDINALITY = u'interana_excluded'
For fields that should not be indexed due to high cardinality (e.g. not used in Interana)

class baseplate.events.**Event**(*topic*, *event_type*, *timestamp=None*, *id=None*)
An event.

get_field(*key*)
Get the value of a field in the event.

If the field is not present, `None` is returned.

Parameters **key** (*str*) – The name of the field.

set_field(*key*, *value*, *obfuscate=False*, *kind=<FieldKind.NORMAL: None>*)
Set the value for a field in the event.

Parameters

- **key** (*str*) – The name of the field.
- **value** – The value to set the field to. Should be JSON serializable.
- **kind** (`baseplate.events.FieldKind`) – The kind the field is. Used to determine what section of the payload the field belongs in when serialized.

2.4.2 Queing Events

class baseplate.events.**EventQueue**(*name*, *event_serializer=<function serialize_v1_event>*)
A queue to transfer events to the publisher.

Parameters

- **name** (*str*) – The name of the event queue to send to. This specifies which publisher should send the events which can be useful for routing to different event pipelines (prod/test/v2 etc.).
- **event_serializer** (*callable*) – A callable that takes an event object and returns serialized bytes ready to send on the wire. See below for options.

put(*event*)
Add an event to the queue.

The queue is local to the server this code is run on. The event publisher on the server will take these events and send them to the collector.

Parameters **event** – The event to send. The type of event object passed in depends on the selected `event_serializer`.

Raises *EventTooLargeError* The serialized event is too large.

Raises *EventQueueFullError* The queue is full. Events are not being published fast enough.

The `EventQueue` also implements *ContextFactory* so it can be used with *add_to_context()*:

```
event_queue = EventQueue("production")
baseplate.add_to_context("events_production", event_queue)
```

It can then be used from the *context object* during requests:

```
def some_service_method(self, context):
    event = Event(...)
    context.events_production.put(event)
```

Serializers

The `event_serializer` parameter to *EventQueue* is a callable which serializes a given event object. The default is the original schemaless format. This can be overridden by passing in a different serializer. Baseplate comes with a serializer for the new Thrift schema based V2 event system as well:

baseplate.events.**serialize_v1_event**(*event*)
Serialize an Event object for the V1 event protocol.

Parameters **event** (baseplate.events.Event) – An event object.

baseplate.events.**serialize_v2_event**(*event*)
Serialize a Thrift struct to bytes for the V2 event protocol.

Parameters **event** – A Thrift struct from the event schemas.

Exceptions

exception baseplate.events.**EventError**
Base class for event related exceptions.

exception baseplate.events.**EventTooLargeError**(*size*)
Raised when a serialized event is too large to send.

exception baseplate.events.**EventQueueFullError**
Raised when the queue of events is full.

This usually indicates that the event publisher is having trouble talking to the event collector.

2.4.3 Publishing Events

Events that are put onto an *EventQueue* are consumed by a separate process and published to the remote event collector service. The publisher is in baseplate and can be run as follows:

```
python -m baseplate.events.publisher --queue-name something config_file.ini
```

The publisher will look at the specified INI file to find its configuration. Given a queue name of `something` (as in the example above), it will expect a section in the INI file called `[event-publisher:something]` with content like below:

```
[event-publisher:something]
collector.hostname = some-domain.example.com

key.name = NameOfASecretKey
key.secret = Base64-encoded-blob-of-randomness

metrics.namespace = a.name.to.put.metrics.under
metrics.endpoint = the-statsd-host:1234
```

2.5 baseplate.experiments

2.5.1 Experiment Providers

baseplate.experiments.providers.r2

Classes

class baseplate.experiments.providers.r2.**R2Experiment**(*id*, *name*, *owner*, *variants*, *seed=None*, *bucket_val=u'user_id'*, *targeting=None*, *overrides=None*, *newer_than=None*, *version=None*)

A "legacy", r2-style experiment.

Deprecated since version 0.27: Use SimpleExperiment with SingleVariantSet or MultiVariantSet instead.

Should log bucketing events to the event pipeline.

Note that this style of experiment caps the size of your variants such that:

```
def max_variant_size(variant_size, num_variants):
    return max(variant_size, (1/num_variants) * 100)
```

The config dict is expected to have the following values:

- **variants**: dict mapping variant names to their sizes. Variant sizes are expressed as numeric percentages rather than a fraction of 1 (that is, 1.5 means 1.5%, not 150%).
- **targeting**: (Optional) dict that maps the names of targeting parameters to lists of valid values. When determining the variant of an experiment, the targeting parameters you want to use are passed in as keyword arguments to the call to experiment.variant.
- **overrides**: (Optional) dict that maps override parameters to dictionaries mapping values to the variant name you want to override the variant to. When determining the variant of an experiment, the override parameters you want to use are passed in as keyword arguments to the call to experiment.variant.
- **bucket_val**: (Optional) Name of the parameter you want to use for bucketing. This value must be passed to the call to experiment.variant as a keyword argument. Defaults to "user_id".
- **seed**: (Optional) Overrides the seed for this experiment. If this is not set, *name* is used as the seed.
- **newer_than**: (Optional) The earliest time that a bucketing resource can have been created by in UTC epoch seconds. If set, you must pass the time, in UTC epoch seconds, when the resource that you are bucketing was created to the call to experiment.variant as the "created" parameter. For example, if you are bucketing based on user_id, created would be set to the time when a User account was created or when an LoID cookie was generated.

baseplate.experiments.providers.feature_flag

Classes

class baseplate.experiments.providers.feature_flag.**FeatureFlag**(*id*, *name*, *owner*, *variants*, *seed=None*, *bucket_val=u'user_id'*, *targeting=None*, *overrides=None*, *newer_than=None*, *version=None*)

An experiment with a single variant "active".

Deprecated since version 0.27: Use SimpleExperiment with RolloutVariantSet instead.

Does not log bucketing events to the event pipeline. Use this type of experiment if you just want to control access to a feature but do not want to run an actual experiment. Some examples for when you would want to use a FeatureFlag are:

1. Slowly rolling out a new feature to a % of users
2. Restricting a new feature to certain subreddits

The config dict is expected to have the following values:

- **variants**: dict mapping variant names to their sizes. Variant sizes are expressed as numeric percentages rather than a fraction of 1 (that is, 1.5 means 1.5%, not 150%). For a feature flag, you can only specify a single variant named "active".
- **targeting**: (Optional) dict that maps the names of targeting parameters to lists of valid values. When determining the variant of an experiment, the targeting parameters you want to use are passed in as keyword arguments to the call to experiment.variant.
- **overrides**: (Optional) dict that maps override parameters to dictionaries mapping values to the variant name you want to override the variant to. When determining the variant of an experiment, the override parameters you want to use are passed in as keyword arguments to the call to experiment.variant.
- **bucket_val**: (Optional) Name of the parameter you want to use for bucketing. This value must be passed to the call to experiment.variant as a keyword argument. Defaults to "user_id".
- **seed**: (Optional) Overrides the seed for this experiment. If this is not set, *name* is used as the seed.
- **newer_than**: (Optional) The earliest time that a bucketing resource can have been created by in UTC epoch seconds. If set, you must pass the time, in UTC epoch seconds, when the resource that you are bucketing was created to the call to experiment.variant as the "created" parameter. For example, if you are bucketing based on user_id, created would be set to the time when a User account was created or when an LoID cookie was generated.

baseplate.experiments.providers.forced_variant

Classes

class baseplate.experiments.providers.forced_variant.**ForcedVariantExperiment**(*variant*)

An experiment that always returns a specified variant.

Deprecated since version 0.27.

Should not log bucketing events to the event pipeline. Note that ForcedVariantExperiments are not directly configured, rather they are used when an experiment is disabled or when "global_override" is set in the base config.

baseplate.experiments.providers.simple_experiment

Classes

class baseplate.experiments.providers.simple_experiment.**SimpleExperiment**(*id*, *name*, *owner*, *start_ts*, *stop_ts*, *config*, *experiment_version*, *shuffle_version*, *variant_set*, *bucket_seed*, *bucket_val*, *targeting*, *overrides*, *enabled=True*, *log_bucketing=True*, *num_buckets=1000*)

A basic experiment choosing from a set of variants.

Simple experiments are meant to be used in conjunction with a VariantSet. This class serves as the replacement for the legacy r2 and feature_flag providers.

2.5.2 Configuration Parsing

baseplate.experiments.**experiments_client_from_config**(*app_config*, *event_logger*)

Configure and return an *ExperimentsContextFactory* object.

This expects one configuration option:

experiments.path The path to the experiment config file generated by the experiment config fetcher daemon.

Parameters

- **raw_config** (*dict*) – The application configuration which should have settings for the experiments client.

- **event_logger** (*baseplate.events.EventLogger*) – The EventLogger to be used to log bucketing events.

Return type *ExperimentsContextFactory*

baseplate.experiments.providers.**parse_experiment**(*config*)

Factory method that parses an experiment config dict and returns an appropriate Experiment class.

The config dict is expected to have the following values:

- **id**: Integer experiment ID, should be unique for each experiment.
- **name**: String experiment name, should be unique for each experiment.
- **owner**: The group or individual that owns this experiment.
- **version**: String to identify the specific version of the experiment.
- **start_ts**: A float of seconds since the epoch of date and time when you want the experiment to start. If an experiment has not been started yet, it is considered disabled.
- **stop_ts**: A float of seconds since the epoch of date and time when you want the experiment to stop. Once an experiment is stopped, it is considered disabled.
- **type**: String specifying the type of experiment to run. If this value is not recognized, the experiment will be considered disabled.
- **experiment**: The experiment config dict for the specific type of experiment. The format of this is determined by the specific experiment type.
- **enabled**: (Optional) If set to False, the experiment will be disabled and calls to experiment.variant will always return None and will not log bucketing events to the event pipeline. Defaults to True.
- **global_override**: (Optional) If this is set, calls to experiment.variant will always return the override value and will not log bucketing events to the event pipeline.

Parameters **config** (*dict*) – Configuration dict for the experiment you wish to run.

Return type baseplate.experiments.providers.base.Experiment

Returns A subclass of `Experiment` for the given experiment type.

2.5.3 Classes

class baseplate.experiments.**ExperimentsContextFactory**(*path*, *event_logger=None*, *timeout=None*)

Experiment client context factory

This factory will attach a new *baseplate.experiments.Experiments* to an attribute on the *context object*.

class baseplate.experiments.**Experiments**(*config_watcher*, *server_span*, *context_name*, *event_logger=None*)

Access to experiments with automatic refresh when changed.

This experiments client allows access to the experiments cached on disk by the experiment config fetcher daemon. It will automatically reload the cache when changed. This client also handles logging bucketing events to the event pipeline when it is determined that the request is part of an active variant.

get_all_experiment_names()

Return a list of all valid experiment names from the configuration file.

Return type list

Returns List of all valid experiment names.

is_valid_experiment(*name*)
Return true if the provided experiment name is a valid experiment.

Parameters **name** (*str*) – Name of the experiment you want to check.

Return type bool

Returns Whether or not a particular experiment is valid.

variant(*name*, *user=None*, *bucketing_event_override=None*, ***kwargs*)
Return which variant, if any, is active.

If a variant is active, a bucketing event will be logged to the event pipeline unless any one of the following conditions are met:

1. bucketing_event_override is set to False.
2. The experiment specified by "name" explicitly disables bucketing events.
3. We have already logged a bucketing event for the value specified by experiment.get_unique_id(**kwargs) within the current request.

Since checking the status an experiment will fire a bucketing event, it is best to only check the variant when you are making the decision that will expose the experiment to the user. If you absolutely must check the status of an experiment before you are sure that the experiment will be exposed to the user, you can use *bucketing_event_override* to disabled bucketing events for that check.

Parameters

- **name** (*str*) – Name of the experiment you want to run.
- **user** (baseplate.core.User) – (Optional) User object for the user you want to check the experiment variant for. If you set user, the experiment parameters for that user ("user_id", "logged_in", and "user_roles") will be extracted and added to the inputs to the call to Experiment.variant. The user's event_fields will also be extracted and added to the bucketing event if one is logged. It is recommended that you provide a value for user rather than setting the user parameters manually in kwargs.
- **bucketing_event_override** (*bool*) – (Optional) Set if you need to override the default behavior for sending bucketing events. This parameter should be set sparingly as it breaks the assumption that you will fire a bucketing event when you first check the state of an experiment. If set to False, will never send a bucketing event. If set to None, no override will be applied. Set to None by default. Note that setting bucketing_event_override to True has no effect, it will behave the same as when it is set to None.
- **kwargs** – Arguments that will be passed to experiment.variant to determine bucketing, targeting, and overrides. These values will also be passed to the logger.

Return type str

Returns Variant name if a variant is active, None otherwise.

expose(*experiment_name*, *variant_name*, *user=None*, ***kwargs*)
Log an event to indicate that a user has been exposed to an experimental treatment.

Parameters

- **experiment_name** (*str*) – Name of the experiment that was exposed.
- **variant_name** (*str*) – Name of the variant that was exposed.

- **user** (baseplate.core.User) – (Optional) User object for the user you want to check the experiment variant for. If unset, it is expected that user_id and logged_in values will be set in the kwargs
- **kwargs** – Additional arguments that will be passed to logger.

2.6 baseplate.file_watcher

Watch a file and keep a parsed copy in memory that's updated on changes.

The contents of the file are re-loaded and parsed only when necessary.

For example, a JSON file like the following:

```
{
  "one": 1,
  "two": 2
}
```

might be watched and parsed like this:

```
>>> watcher = FileWatcher(path, parser=json.load)
>>> watcher.get_data() == {u"one": 1, u"two": 2}
True
```

The return value of *get_data()* would change whenever the underlying file changes.

class baseplate.file_watcher.**FileWatcher**(*path*, *parser*, *timeout=None*)

Watch a file and load its data when it changes.

Parameters

- **path** (*str*) – Full path to a file to watch.
- **parser** (*callable*) – A callable that takes an open file object, parses or otherwise interprets the file, and returns whatever data is meaningful.
- **timeout** (*float*) – How long, in seconds, to block instantiation waiting for the watched file to become available (defaults to not blocking).

get_data()

Return the current contents of the file, parsed.

The watcher ensures that the file is re-loaded and parsed whenever its contents change. Parsing only occurs when necessary, not on each call to this method. This method returns whatever the most recent call to the parser returned.

Make sure to call this method each time you need data from the file rather than saving its results elsewhere. This ensures you always have the freshest data.

2.6.1 Exceptions

exception baseplate.file_watcher.**WatchedFileNotAvailableError**(*path*, *inner*)

Raised when the watched file could not be loaded.

2.7 baseplate.live_data

This component of Baseplate provides real-time synchronization of data across a cluster of servers. It is intended for situations where data is read frequently, doesn't change super often, and when it does change needs to change everywhere at once. In most cases, this will be an underlying feature of some other system (e.g. an experiments framework.)

There are four main components of the live data system:

- ZooKeeper, a highly available data store that can push change notifications.
- The watcher, a sidecar daemon that watches nodes in ZooKeeper and syncs their contents to disk.
- *FileWatcher* instances in your application that load the synchronized data into memory.
- Something that writes to ZooKeeper (potentially the writer tool).

The watcher daemon and tools for writing data to ZooKeeper are covered on this page.

2.7.1 Watcher Daemon

The watcher daemon is a sidecar that watches nodes in ZooKeeper and syncs their contents to local files on change. It is entirely configured via INI file and is run like so:

```
$ python -m baseplate.live_data.watcher some_config.ini
```

Where `some_config.ini` might look like:

```
[live-data]
zookeeper.hosts = zk01:2181,zk02:2181
zookeeper.credentials = secret/myservice/zookeeper_credentials

nodes.a.source = /a/node/in/zookeeper
nodes.a.dest = /var/local/file-on-disk

nodes.b.source = /another/node/in/zookeeper
nodes.b.dest = /var/local/another-file
nodes.b.owner = www-data
nodes.b.group = www-data
nodes.b.mode = 0400
```

Each of the defined `nodes` will be watched by the daemon.

The watcher daemon will touch the `mtime` of the local files periodically to indicative liveliness to monitoring tools.

2.7.2 The Writer Tool

For simple cases where you just want to put the contents of a file into ZooKeeper (perhaps in a CI task) you can use the live data writer. It expects a configuration file with ZooKeeper connection information, like the watcher, and takes some additional parameters on the command line.

```
$ python -m baseplate.live_data.writer some_config.ini \
   input.json /some/node/in/zookeeper
Writing input.json to ZooKeeper /some/node/in/zookeeper...
---
```

(continues on next page)

(continued from previous page)

```
+++

@@ -1,4 +1,4 @@

{
-    "key": "one"
+    "key": "two"
}
Wrote data to Zookeeper.
```

The ZooKeeper node must be created before this tool can be used so that appropriate ACLs can be configured.

2.7.3 Direct access to ZooKeeper

If you're doing something more complicated with your data that the above tools don't cover, you'll want to connect directly to ZooKeeper.

2.8 baseplate.message_queue

This module provides a thin wrapper around POSIX Message queues.

Note: This implementation uses POSIX Message queues and is not portable to all operating systems.

There are also various limits on the sizes of queues:

- The `msgqueue rlimit` limits the amount of space the user can use on message queues.
- The `fs.mqueue.msg_max` and `fs.mqueue.msgsize_max` sysctls limit the maximum number of messages and the maximum size of each message which a queue can be configured to have.

2.8.1 Minimal Example

Here's a minimal, artificial example of a separate producer and consumer process pair (run the producer then the consumer):

```
# producer.py
from baseplate.message_queue import MessageQueue

# If the queue doesn't already exist, we'll create it.
mq = MessageQueue(
    "/baseplate-testing", max_messages=1, max_message_size=1)
message = "1"
mq.put(message)
print("Put Message: %s" % message)
```

You should see:

```
Put Message: 1
```

After running the producer once, we have a single message pushed on to our POSIX message queue. Next up, run the consumer:

```
# consumer.py
from baseplate.message_queue import MessageQueue

mq = MessageQueue(
    "/baseplate-testing", max_messages=1, max_message_size=1)
# Unless a `timeout` kwarg is passed, this will block until
# we can pop a message from the queue.
message = mq.get()
print("Get Message: %s" % message)
```

You'll end up seeing:

```
Get Message: 1
```

The `/baseplate-testing` value is the name of the queue. Queues names should start with a forward slash, followed by one or more characters (but no additional slashes).

Multiple processes can bind to the same queue by specifying the same queue name.

2.8.2 Message Queue Default Limits

Most operating systems with POSIX queues include very low defaults for the maximum message size and maximum queue depths. On Linux 2.6+, you can list and check the values for these by running:

```
$ ls /proc/sys/fs/mqueue/
msg_default  msg_max  msgsize_default  msgsize_max  queues_max
$ cat /proc/sys/fs/mqueue/msgsize_max
8192
```

Explaining these in detail is outside the scope of this document, so we'll refer you to POSIX Message queues (or `man 7 mq_overview`) for detailed instructions on what these mean.

2.8.3 Gotchas

If you attempt to create a POSIX Queue where one of your provided values is over the limits defined under `/proc/sys/fs/mqueue/`, you'll probably end up seeing a vague `ValueError` exception. Here's an example:

```
>>> from baseplate.message_queue import MessageQueue
>>> mq = MessageQueue(
        "/over-the-limit", max_messages=11, max_message_size=8096)
Traceback (most recent call last):
  File "<input>", line 2, in <module>
  File "/home/myuser/baseplate/baseplate/message_queue.py", line 83, in __init__
    max_message_size=max_message_size,
ValueError: Invalid parameter(s)
```

Since the default value for `/proc/sys/fs/mqueue/msg_max` on Linux is 10, our `max_messages=11` is invalid. You can raise these limits by doing something like this as a privileged user:

```
$ echo "50" > /proc/sys/fs/mqueue/msg_max
```

2.8.4 CLI Usage

The *message_queue* module can also be run as a command-line tool to consume, log, and discard messages from a given queue:

```
python -m baseplate.message_queue --read /queue
```

or to write arbitrary messages to the queue:

```
echo hello! | python -m baseplate.message_queue --write /queue
```

See `--help` for more info.

2.8.5 baseplate.message_queue

A gevent-friendly POSIX message queue.

class baseplate.message_queue.**MessageQueue**(*name*, *max_messages*, *max_message_size*)
A gevent-friendly (but not required) inter process message queue.

`name` should be a string of up to 255 characters consisting of an initial slash, followed by one or more characters, none of which are slashes.

Note: This relies on POSIX message queues being available and select(2)-able like other file descriptors. Not all operating systems support this.

get(*timeout=None*)
Read a message from the queue.

Parameters **timeout** (*float*) – If the queue is empty, the call will block up to `timeout` seconds or forever if `None`.

Raises *TimedOutError* The queue was empty for the allowed duration of the call.

put(*message*, *timeout=None*)
Add a message to the queue.

Parameters **timeout** (*float*) – If the queue is full, the call will block up to `timeout` seconds or forever if `None`.

Raises *TimedOutError* The queue was full for the allowed duration of the call.

unlink()
Remove the queue from the system.

The queue will not leave until the last active user closes it.

close()
Close the queue, freeing related resources.

This must be called explicitly if queues are created/destroyed on the fly. It is not automatically called when the object is reclaimed by Python.

2.8.6 Exceptions

exception baseplate.message_queue.**MessageQueueError**
Base exception for message queue related errors.

exception baseplate.message_queue.**TimedOutError**
Raised when a message queue operation times out.

2.9 baseplate.metrics

Application metrics via statsd.

A client for the application metrics aggregator statsd. Metrics sent to statsd are aggregated and written to graphite. Statsd is generally used for whole-system health monitoring and insight into usage patterns.

Basic example usage:

```
from baseplate import metrics_client_from_config

client = metrics_client_from_config(app_config)
client.counter("events.connect").increment()
client.gauge("workers").replace(4)

with client.timer("something.todo"):
    do_something()
    do_something_else()
```

If you have multiple metrics to send, you can batch them up for efficiency:

```
with client.batch() as batch:
    batch.counter("froozles").increment()
    batch.counter("blargs").decrement(delta=3)

    with batch.timer("something"):
        do_another_thing()
```

and the batch will be sent in as few packets as possible when the *with* block ends.

2.9.1 Clients

baseplate.metrics.**make_client**(*namespace*, *endpoint*)

Return a configured client.

Parameters

- **namespace** (*str*) – The root key to prefix all metrics with.
- **endpoint** (baseplate.config.EndpointConfiguration) – The endpoint to send metrics to or None. If None, the returned client will discard all metrics.

Returns A configured client.

Return type *baseplate.metrics.Client*

See also:

baseplate.metrics_client_from_config().

class baseplate.metrics.**Client**

A client for statsd.

batch()

Return a client-like object which batches up metrics.

Batching metrics can reduce the number of packets that are sent to the stats aggregator.

Return type *Batch*

counter(*name*)
Return a Counter with the given name.

The sample rate is currently up to your application to enforce.

Parameters **name** (*str*) – The name the counter should have.

Return type *Counter*

gauge(*name*)
Return a Gauge with the given name.

Parameters **name** (*str*) – The name the gauge should have.

Return type *Gauge*

histogram(*name*)
Return a Histogram with the given name.

Parameters **name** (*str*) – The name the histogram should have.

Return type *Histogram*

timer(*name*)
Return a Timer with the given name.

Parameters **name** (*str*) – The name the timer should have.

Return type *Timer*

class baseplate.metrics.**Batch**
A batch of metrics to send to statsd.

The batch also supports the context manager protocol, for use with Python's `with` statement. When the context is exited, the batch will automatically *flush()*.

flush()
Immediately send the batched metrics.

counter(*name*)
Return a BatchCounter with the given name.

The sample rate is currently up to your application to enforce.

Parameters **name** (*str*) – The name the counter should have.

Return type *Counter*

gauge(*name*)
Return a Gauge with the given name.

Parameters **name** (*str*) – The name the gauge should have.

Return type *Gauge*

histogram(*name*)
Return a Histogram with the given name.

Parameters **name** (*str*) – The name the histogram should have.

Return type *Histogram*

timer(*name*)
Return a Timer with the given name.

Parameters **name** (*str*) – The name the timer should have.

Return type *Timer*

2.9.2 Metrics

class baseplate.metrics.**Counter**
A counter for counting events over time.

increment(*delta=1*, *sample_rate=1.0*)
Increment the counter.

Parameters

- **delta** (*float*) – The amount to increase the counter by.
- **sample_rate** (*float*) – What rate this counter is sampled at. [0-1].

decrement(*delta=1*, *sample_rate=1.0*)
Decrement the counter.

This is equivalent to *increment()* with delta negated.

send(*delta*, *sample_rate*)
Send the counter to the backend.

Parameters

- **delta** (*float*) – The amount to increase the counter by.
- **sample_rate** (*float*) – What rate this counter is sampled at. [0-1].

class baseplate.metrics.**Timer**
A timer for recording elapsed times.

The timer also supports the context manager protocol, for use with Python's `with` statement. When the context is entered the timer will *start()* and when exited, the timer will automatically *stop()*.

start()
Record the current time as the start of the timer.

stop()
Stop the timer and record the total elapsed time.

class baseplate.metrics.**Gauge**
A gauge representing an arbitrary value.

Note: The statsd protocol supports incrementing/decrementing gauges from their current value. We do not support that here because this feature is unpredictable in face of the statsd server restarting and the "current value" being lost.

---

replace(*new_value*)
Replace the value held by the gauge.

This will replace the value held by the gauge with no concern for its previous value.

Note: Due to the way the protocol works, it is not possible to replace gauge values with negative numbers.

Parameters **new_value** (*float*) – The new value to store in the gauge.

class baseplate.metrics.**Histogram**
A bucketed distribution of integer values across a specific range.

Records data value counts across a configurable integer value range with configurable buckets of value precision within that range.

Configuration of each histogram is managed by the backend service, not by this interface. This implementation also depends on histograms being supported by the StatsD backend. Specifically, the StatsD backend must support the `h` key, e.g. `metric_name:320|h`.

add_sample(*value*)

Add a new value to the histogram.

This records a new value to the histogram; the bucket it goes in is determined by the backend service configurations.

2.10 baseplate.queue_consumer

Create a long-running process to consume from a queue. For example:

```
from kombu import Connection, Exchange
from baseplate import queue_consumer

def process_links(context, msg_body, msg):
    print('processing %s' % msg_body)

queue_consumer.consume(
    baseplate=make_baseplate(cfg, app_config),
    exchange=Exchange('reddit_exchange', 'direct'),
    connection=Connection(
      hostname='amqp://guest:guest@reddit.local:5672',
      virtual_host='/',
    ),
    queue_name='process_links_q',
    routing_keys=[
        'link_created',
        'link_deleted',
        'link_updated',
    ],
    handler=process_links,
)
```

This will create a queue named `'process_links_q'` and bind the routing keys `'link_created'`, `'link_deleted'`, and `'link_updated'`. It will then register a consumer for `'process_links_q'` to read messages and feed them to `process_links`.

2.10.1 Register and run a queue consumer

baseplate.queue_consumer.**consume**(*baseplate*, *exchange*, *connection*, *queue_name*, *routing_keys*, *handler*)

Create a long-running process to consume messages from a queue.

A queue with name `queue_name` is created and bound to the `routing_keys` so messages published to the `routing_keys` are routed to the queue.

Next, the process registers a consumer that receives messages from the queue and feeds them to the `handler`.

The `handler` function must take 3 arguments:

- `context`: a baseplate context

- `message_body`: the text body of the message
- `message`: `kombu.message.Message`

The consumer will automatically `ack` each message after the handler method exits. If there is an error in processing and the message must be retried the handler should raise an exception to crash the process. This will prevent the `ack` and the message will be re-queued at the head of the queue.

Parameters

- **baseplate** (`baseplate.core.Baseplate`) – A baseplate instance for the service.
- **exchange** (*kombu.Exchange*) –
- **connection** (*kombu.connection.Connection*) –
- **queue_name** (*str*) – The name of the queue.
- **routing_keys** (*list*) – List of routing keys.
- **handler** – The handler method.

class baseplate.queue_consumer.**KombuConsumer**(*worker*, *worker_thread*)
Consumer for use in baseplate.

The *get_message()* and *get_batch()* methods will automatically record diagnostic information.

get_message(*server_span*)
Return a single message.

Parameters **server_span** (`baseplate.core.ServerSpan`) –

get_batch(*server_span*, *max_items*, *timeout*)
Return a batch of messages.

Parameters

- **server_span** (`baseplate.core.ServerSpan`) –
- **max_items** (*int*) – The maximum batch size.
- **timeout** (*int*) – The maximum time to wait in seconds, or `None` for no timeout.

2.10.2 If you require more direct control

class baseplate.queue_consumer.**BaseKombuConsumer**(*worker*, *worker_thread*)
Base object for consuming messages from a queue.

A worker process accepts messages from the queue and puts them in a local work queue. The "real" consumer can then get messages with *get_message()* or *get_batch()*. It is that consumer's responsibility to `ack` or `reject` messages.

Can be used directly, outside of standard baseplate context.

classmethod new(*connection*, *queues*)
Create and initialize a consumer.

Parameters

- **exchange** (*kombu.Exchange*) –
- **queues** (*list*) – List of `kombu.queue.Queue` objects.

get_message()
Return a single message.

get_batch(*max_items*, *timeout*)
Return a batch of messages.

Parameters

- **max_items** (*int*) – The maximum batch size.
- **timeout** (*int*) – The maximum time to wait in seconds, or `None` for no timeout.

2.11 baseplate.random

Extensions to the standard library *random* module.

class baseplate.random.**WeightedLottery**(*items*, *weight_key*)
A lottery where items can have different chances of selection.

Items will be picked with chance proportional to their weight relative to the sum of all weights, so the higher the weight, the higher the chance of being picked.

Parameters

- **items** – A sequence of items to choose from.
- **weight_key** – A function that takes an item in `items` and returns a non-negative integer weight for that item.

Raises `ValueError` if any weights are negative or there are no items.

An example of usage:

```
>>> words = ["apple", "banana", "cantelope"]
>>> lottery = WeightedLottery(words, weight_key=len)
>>> lottery.pick()
'banana'
>>> lottery.sample(2)
['apple', 'cantelope']
```

pick()
Pick a random element from the lottery.

sample(*sample_size*)
Sample elements from the lottery without replacement.

Parameters **sample_size** (*int*) – The number of items to sample from the lottery.

2.12 baseplate.retry

Note: This module is a low-level helper, many client libraries have protocol-aware retry logic built in. Check your library before using this.

Policies for retrying an operation safely.

class baseplate.retry.**RetryPolicy**
A policy for retrying operations.

Policies are meant to be used as an iterable:

```
for time_remaining in RetryPolicy.new(attempts=3):
    try:
        some_operation.do(timeout=time_remaining)
        break
    except SomeError:
        pass
else:
    raise MaxRetriesError
```

yield_attempts()
Return an iterator which controls attempts.

On each iteration, the iterator will yield the number of seconds left to retry, this should be used to set the timeout on the operation being carried out. If there is no maximum time remaining, `None` is yielded instead.

The iterable will raise `StopIteration` once the operation should not be retried any further.

__iter__()
Convenience alias for *yield_attempts()*.

This allows policies to be directly iterated over.

static new(*attempts=None*, *budget=None*, *backoff=None*)
Create a new retry policy with the given constraints.

Parameters

- **attempts** (*int*) – The maximum number of times the operation can be attempted.
- **budget** (*float*) – The maximum amount of time, in seconds, that the local service will wait for the operation to succeed.
- **backoff** (*float*) – The base amount of time, in seconds, for exponential backoff between attempts. `N` in (`N * 2**attempts`).

2.13 baseplate.secrets

Secure access to secret tokens stored in Vault.

2.13.1 Fetcher Daemon

The secret fetcher is a sidecar that is run as a single daemon on each server. It can authenticate to Vault either as the server itself (through an AWS-signed instance identity document) or through a mounted JWT when running within a Kubernetes pod. It then gets access to secrets based upon the policies mapped to the role it authenticated as. Once authenticated, it fetches a given list of secrets from Vault and stores all of the data in a local file. It will automatically re-fetch secrets as their leases expire, ensuring that key rotation happens on schedule.

Because this is a sidecar, individual application processes don't need to talk directly to Vault for simple secret tokens (but can do so if needed for more complex operations like using the Transit backend). This reduces the load on Vault and adds a safety net if Vault becomes unavailable.

2.13.2 Secret Store

The secret store is the in-application integration with the file output of the fetcher daemon.

baseplate.secrets.**secrets_store_from_config**(*app_config*, *timeout=None*)
Configure and return a secrets store.

This expects one configuration option:

secrets.path The path to the secrets file generated by the secrets fetcher daemon.

Parameters

- **raw_config** (*dict*) – The application configuration which should have settings for the secrets store.
- **timeout** (*float*) – How long, in seconds, to block instantiation waiting for the secrets data to become available (defaults to not blocking).

Return type *SecretsStore*

class baseplate.secrets.**SecretsStore**(*path*, *timeout=None*)
Access to secret tokens with automatic refresh when changed.

This local vault allows access to the secrets cached on disk by the fetcher daemon. It will automatically reload the cache when it is changed. Do not cache or store the values returned by this class's methods but rather get them from this class each time you need them. The secrets are served from memory so there's little performance impact to doing so and you will be sure to always have the current version in the face of key rotation etc.

get_raw(*path*)
Return a dictionary of key/value pairs for the given secret path.

This is the raw representation of the secret in the underlying store.

Return type dict

get_simple(*path*)
Decode and return a simple secret.

Simple secrets are a convention of key/value pairs in the raw secret payload. The following keys are significant:

type This must always be `simple` for this method.

value This contains the raw value of the secret token.

encoding (Optional) If present, how to decode the value from how it's encoded at rest (only `base64` currently supported).

Return type bytes

get_versioned(*path*)
Decode and return a versioned secret.

Versioned secrets are a convention of key/value pairs in the raw secret payload. The following keys are significant:

type This must always be `versioned` for this method.

current, next, and previous The raw secret value's versions. `current` is the "active" version, which is used for new creation/signing operations. `previous` and `next` are only used for validation (e.g. checking signatures) to ensure continuity when keys rotate. Both `previous` and `next` are optional.

encoding (Optional) If present, how to decode the values from how they are encoded at rest (only `base64` currently supported).

Return type *VersionedSecret*

get_vault_url()
Return the URL for accessing Vault directly.

Return type str

See also:

The *baseplate.context.hvac* module provides integration with HVAC, a Vault client.

get_vault_token()
Return a Vault authentication token.

The token will have policies attached based on the current EC2 server's Vault role. This is only necessary if talking directly to Vault.

Return type str

See also:

The *baseplate.context.hvac* module provides integration with HVAC, a Vault client.

make_object_for_context(*name*, *server_span*)
Return an object that can be added to the context object.

This allows the secret store to be used with *add_to_context()*:

```
secrets = SecretsStore("/var/local/secrets.json")
baseplate.add_to_context("secrets", secrets)
```

class baseplate.secrets.**VersionedSecret**
A versioned secret.

Versioned secrets allow for seamless rotation of keys. When using the secret to generate tokens (e.g. signing a message) always use the `current` value. When validating tokens, check against all the versions in `all_versions`. This will allow keys to rotate smoothly even if not done instantly across all users of the secret.

all_versions
Return an iterator over the available versions of this secret.

classmethod from_simple_secret(*value*)
Make a fake versioned secret from a single value.

This is a backwards compatibility shim for use with APIs that take versioned secrets. Try to use proper versioned secrets fetched from the secrets store instead.

Exceptions

exception baseplate.secrets.**CorruptSecretError**(*path*, *message*)
Raised when the requested secret does not match the expected format.

exception baseplate.secrets.**SecretNotFoundError**(*name*)
Raised when the requested secret is not in the local vault.

exception baseplate.secrets.**SecretsNotAvailableError**(*inner*)
Raised when the secrets store was not accessible.

2.14 baseplate.thrift_pool

A Thrift client connection pool.

Note: See *baseplate.context.thrift.ThriftContextFactory* for a convenient way to integrate the pool with your application.

The pool lazily creates connections and maintains them in a pool. Individual connections have a maximum lifetime, after which they will be recycled.

A basic example of usage:

```
pool = thrift_pool_from_config(app_config, "example_service.")
with pool.connection() as protocol:
    client = ExampleService.Client(protocol)
    client.do_example_thing()
```

2.14.1 Configuration Parsing

baseplate.thrift_pool.**thrift_pool_from_config**(*app_config*, *prefix*, ***kwargs*)

Make a ThriftConnectionPool from a configuration dictionary.

The keys useful to *thrift_pool_from_config()* should be prefixed, e.g. `example_service.endpoint` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the *ThriftConnectionPool* constructor. Any keyword arguments given to this function will be also be passed through to the constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- **`endpoint` (required): A `host:port` pair, e.g. `localhost:2014`,** where the Thrift server can be found.
- `size`: The size of the connection pool.
- **`max_age`: The oldest a connection can be before it's recycled and** replaced with a new one. Written as a time span e.g. `1 minute`.
- **`timeout`: The maximum amount of time a connection attempt or RPC call** can take before a TimeoutError is raised.
- **`max_retries`: The maximum number of times the pool will attempt to** open a connection.

2.14.2 Classes

class baseplate.thrift_pool.**ThriftConnectionPool**

A pool that maintains a queue of open Thrift connections.

Parameters

- **`endpoint`** (*baseplate.config.EndpointConfiguration*) – The remote address of the Thrift service.
- **`size`** (*int*) – The maximum number of connections that can be open before new attempts to open block.

- **max_age** (*int*) – The maximum number of seconds a connection should be kept alive. Connections older than this will be reaped.
- **timeout** (*int*) – The maximum number of seconds a connection attempt or RPC call can take before a TimeoutError is raised.
- **max_retries** (*int*) – The maximum number of times the pool will attempt to open a connection.
- **protocol_factory** (*thrift.protocol.TProtocol.TProtocolBase*) – The factory to use for creating protocols from transports. This is useful for talking to services that don't support THeaderProtocol.

All exceptions raised by this class derive from `TTransportException`.

connection(***kwds*)

Acquire a connection from the pool.

This method is to be used with a context manager. It returns a connection from the pool, or blocks up to `timeout` seconds waiting for one if the pool is full and all connections are in use.

When the context is exited, the connection is returned to the pool. However, if it was exited via an unexpected Thrift exception, the connection is closed instead because the state of the connection is unknown.

2.15 baseplate.service_discovery

Integration with Synapse's `file_output` service discovery method.

Note: Production Baseplate services have Synapse hooked up to a local HAProxy instance which will automatically route connections to services for you if you connect to the correct address/port on localhost. That is the preferred method of connecting to services.

The contents of this module are useful for inspecting the service inventory directly for cases where a blind TCP connection is insufficient (e.g. to give service addresses to a client, or for topology-aware clients like Cassandra).

A basic example of usage:

```
inventory = ServiceInventory("/var/lib/synapse/example.json")
backend = inventory.get_backend()
print(backend.endpoint.address)
```

class baseplate.service_discovery.**ServiceInventory**(*filename*)

The inventory enumerates available backends for a single service.

Parameters **filename** (*str*) – The absolute path to the Synapse-generated inventory file in JSON format.

get_backends()

Return a list of all available backends in the inventory.

If the inventory file becomes unavailable, the previously seen inventory is returned.

Return type list of *Backend* objects

get_backend()

Return a randomly chosen backend from the available backends.

If weights are specified in the inventory, they will be respected when making the random selection.

Return type *Backend*

Raises *NoBackendsAvailableError* if the inventory has no available endpoints.

class baseplate.service_discovery.**Backend**
A description of a service backend.

This is a tuple of several values:

id A unique integer ID identifying the backend.

name The name of the backend.

endpoint An *EndpointConfiguration* object describing the network address of the backend.

weight An integer weight indicating how much to prefer this backend when choosing whom to connect to.

2.15.1 Exceptions

exception baseplate.service_discovery.**NoBackendsAvailableError**
Raised when no backends are available for this service.

CHAPTER 3

The CLI Toolkit

Baseplate provides command line tools which are useful for running applications in production and development.

3.1 `baseplate-healthcheck`

Baseplate services have well-defined health-check endpoints. The `baseplate-healthcheck` tool connects to a given service and checks these endpoints to see if they're alive.

3.1.1 Command Line

There are two required arguments on the command line: the protocol of the service to check (`thrift` or `wsgi`) and the endpoint to connect to.

For example, to check a Thrift-based service listening on port 9090:

```
baseplate-healthcheck thrift 127.0.0.1:9090
```

or a WSGI (HTTP) service listening on a UNIX domain socket:

```
baseplate-healthcheck wsgi /run/myservice.sock
```

3.1.2 Results

If the service is healthy, the tool will exit with a status code indicating success (0) and print "OK!". If the service is unhealthy, the tool will exit with a status code indicating failure (1) and print an error message explaining what went wrong.

3.1.3 Usage

This script can be used as part of a process to validate a server after creation, or to check service liveliness for a service discovery system.

3.2 baseplate-serve

Baseplate comes with a simple Gevent-based server for both Thrift and WSGI applications called `baseplate-serve`.

3.2.1 Configuration

There is one required parameter on the command line, the path to an INI-format configuration file. There should be two sections in the file: the `server` section and the `app` section. The section headers look like `server:main` or `app:main` where the part before the `:` is the type of section and the part after is the "name". Baseplate looks for sections named `main` by default but can be overridden with the `--server-name` and `--app-name` options.

3.2.2 The Server

Here's an example of a `server` section:

```
[server:main]
factory = baseplate.server.thrift
stop_timeout = 30
```

The `factory` tells baseplate what code to use to run the server. Baseplate comes with two servers built in:

`baseplate.server.thrift` A Gevent Thrift server.

`baseplate.server.wsgi` A Gevent WSGI server.

Both take two optional configuration values as well:

`max_concurrency` The maximum number of simultaneous clients the server will handle. Unlimited by default.

`stop_timeout` How long, in seconds, to wait for active connections to finish up gracefully when shutting down. By default, the server will shut down immediately.

The WSGI server takes an additional optional parameter:

`handler` A full name of a class which subclasses `gevent.pywsgi.WSGIHandler` for extra functionality.

3.2.3 The Application

And now the real bread and butter, your `app` section:

```
[app:main]
factory = my_app.processor:make_processor
foo = 3
bar = 22
noodles.blah = one, two, three
```

The `app` section also takes a `factory`. This should be the name of a callable in your code which builds and returns your application. The part before the `:` is a Python module. The part after the `:` is the name of a callable object within that module.

The rest of the options in the `app` section of the configuration file get passed as a dictionary to your application callable. You can parse these options with *`baseplate.config`*.

The application factory should return an appropriate object for your server:

Thrift A `TProcessor`.

WSGI A WSGI callable.

3.2.4 Logging

The baseplate server provides a default configuration for the Python standard `logging` system. The root logger will print to `stdout` with a format that includes trace information. The default log level is `INFO` or `DEBUG` if the `--debug` flag is passed to `baseplate-serve`.

If more complex logging configuration is necessary, the configuration file will override the default setup. The configuration format is documented in the standard library.

3.2.5 Automatic reload on source changes

In development, it's useful for the server to restart itself when you change code. You can do this by passing the `--reload` flag to `baseplate-serve`.

This should not be used in production environments.

3.2.6 Einhorn

`baseplate-serve` can run as a worker in Stripe's Einhorn socket manager. This allows Einhorn to handle binding the socket, worker management, rolling restarts, and worker health checks.

Baseplate supports Einhorn's "manual ACK" protocol. Once the application is loaded and ready to serve, Baseplate notifies the Einhorn master process via its command socket.

An example command line:

```
einhorn -m manual -n 4 --bind localhost:9190 \
   baseplate-serve myapp.ini
```

3.2.7 Debug Signal

Applications running under `baseplate-serve` will respond to `SIGUSR1` by printing a stack trace to the logger. This can be useful for debugging deadlocks and other issues.

Note that Einhorn will exit if you send it a `SIGUSR1`. You can instead open up `einhornsh` and instruct the master to send the signal to all workers:

```
$ einhornsh
> signal SIGUSR1
Successfully sent USR1s to 4 processes: [...]
```

3.3 baseplate-script

This command allows you to run a piece of Python code with the application config loaded similarly to baseplate-serve. The command is `baseplate-script`.

3.3.1 Command Line

There are two required arguments on the command line: the path to an INI-format configuration file, and the fully qualified name of a Python function to run.

The function should be specified as a module path, a colon, and a function name. For example, `my_service.models:create_schema`. The function should take a single argument which will be the application's configuration as a dictionary. This is the same as the application factory used by the server.

Just like with `baseplate-serve`, the `app:main` section will be loaded by default. This can be overridden with the `--app-name` option.

3.3.2 Example

Given a configuration file, `printer.ini`:

```
[app:main]
message = Hello!

[app:bizarro]
message = !olleH
```

and a small script, `printer.py`:

```
def run(app_config):
    print(app_config["message"])
```

You can run the script with various configurations:

```
$ baseplate-script printer.ini printer:run
Hello!

$ baseplate-script printer.ini --app-name=bizarro printer:run
!olleH
```

3.4 baseplate-tshell

This command allows you to run an interactive Python shell for a Thrift service with the application config and context loaded. The command is `baseplate-tshell`.

HTTP services can use Pyramid's pshell in order to get an interactive shell.

3.4.1 Command Line

This command requires the path to an INI-format configuration file to run.

Just like with `baseplate-serve`, the `app:main` section will be loaded by default. This can be overridden with the `--app-name` option.

By default, the shell will have variables containing the application and the context exposed. Additional variables can be exposed by providing a `setup` function in the `tshell` section of the configuration file.

3.4.2 Example

Given a configuration file, `example.ini`:

```
[app:main]
factory = baseplate.server.thrift

[tshell]
setup = my_service:tshell_setup
```

and a small setup function, `my_service.py`:

```
def tshell_setup(env, env_banner):
    from my_service import models
    env['models'] = models
    env_banner['models'] = 'Models module'
```

You can begin a shell with the models module exposed:

```
$ tshell example.ini
Baseplate Interactive Shell
Python 2.7.6 (default, Nov 23 2017, 15:49:48)
[GCC 4.8.4]

Available Objects:

  app           This project's app instance
  context       The context for this shell instance's span
  models        Models module
>>>
```

HTTP services can use Pyramid's pshell in order to get an interactive shell.

CHAPTER 4

Appendix

- genindex
- modindex
- *Glossary*

4.1 Glossary

Context Object An object containing per-request state passed into your request handler. The exact form it takes depends on the framework you are using.

Thrift The `context` object passed into handler functions when using a `ContextProcessor`.

Pyramid The `request` object passed into views.

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y