goblin Documentation

Release 2.1.0

David M. Brown

Feb 06, 2018

Contents

1 Releases **3**

2 Requirements **5**

3 Dependencies **7**

4 Installation **9**

4.1 The Basics . . . 9

4.1.1 Using the OGM . . . 11

4.1.1.1 Modeling Graph Elements with `Goblin` . . . 11

4.1.1.2 Using `goblin.properties` . . . 12

4.1.1.3 Creating Elements and Setting Property Values . . . 12

4.1.1.4 Vertex Properties . . . 13

4.1.1.5 Meta-properties . . . 14

4.1.1.6 Saving Elements to the Database Using `Session` . . . 14

4.1.1.7 Writing Custom Gremlin Traversals . . . 15

4.1.2 Using Graph (GLV) . . . 16

4.1.2.1 The Side Effect Interface . . . 17

4.1.3 Using the Driver . . . 17

4.1.3.1 Connecting to a `Cluster` . . . 17

4.1.3.2 Configuring `Cluster` . . . 18

4.1.4 Configuring the `Goblin` App Object . . . 18

4.1.4.1 Special `Goblin` App Configuration . . . 19

4.1.5 Improving Driver Performance . . . 19

4.1.5.1 Use `cython` . . . 19

4.1.5.2 Use `ujson` . . . 20

4.1.5.3 Use `uvloop` . . . 20

4.1.6 Goblin API . . . 20

4.1.6.1 goblin package . . . 20

5 Indices and tables **29**

Python Module Index **31**

Goblin is an asynchronous Python toolkit for the TinkerPop 3 Gremlin Server. In order to leverage Python's support for asynchronous programming paradigms, *Goblin* is implemented using the async/await syntax introduced in Python 3.5, and does not support earlier Python versions. Goblin is built on top of aiogremlin and provides full compatibility with the aiogremlin GLV and driver.

Main features:

- High level asynchronous *Object Graph Mapper* (OGM)
- Integration with the *official gremlin-python Gremlin Language Variant* (GLV) - now provided by aiogremlin
- Native Python support for asynchronous programing including *coroutines*, *iterators*, and *context managers* as specified in PEP 492
- *Asynchronous Python driver* for the Gremlin Server - now provided by aiogremlin

CHAPTER 1

Releases

The latest release of *goblin* is **2.0.0**.

CHAPTER 2

Requirements

- Python 3.5+
- TinkerPop 3.2.4

CHAPTER 3

Dependencies

- aiogremlin 3.2.4
- inflection 0.3.1

CHAPTER 4

Installation

Install using pip:

```
$ pip install goblin
```

4.1 The Basics

OGM

Define custom vertex/edge classes using the provided base *classes*, *properties*, and *data types*:

```
>>> from goblin import element, properties


>>> class Person(element.Vertex):
...     name = properties.Property(properties.String)
...     age = properties.Property(properties.Integer)


>>> class Knows(element.Edge):
...     notes = properties.Property(properties.String, default='N/A')
```

Create a *Goblin App* and register the element classes:

```
>>> import asyncio
>>> from goblin import Goblin

>>> loop = asyncio.get_event_loop()
>>> app = loop.run_until_complete(
...     Goblin.open(loop))
>>> app.register(Person, Knows)
```

Other than user defined properties, elements provide no interface. Use a *Session* object to interact with the database:

```
>>> async def go(app):
...     session = await app.session()
...     leif = Person()
...     leif.name = 'Leif'
...     leif.age = 28
...     jon = Person()
...     jon.name = 'Jonathan'
...     works_with = Knows(leif, jon)
...     session.add(leif, jon, works_with)
...     await session.flush()
...     result = await session.g.E(works_with.id).next()
...     assert result is works_with
...     people = session.traversal(Person)  # element class based traversal source
...     async for person in people:
...         print(person)

>>> loop.run_until_complete(go(app))
<__main__.Person object at ...>
...
```

Note that a *Goblin session* does not necessarily correspond to a Gremlin Server session. Instead, all elements created using a session are 'live' in the sense that if the results of a traversal executed against the session result in different property values for an element, that element will be updated to reflect these changes.

For more information on using the OGM, see the *OGM docs*

Gremlin Language Variant

Generate and submit Gremlin traversals in native Python:

```
>>> from goblin import DriverRemoteConnection  # alias for aiogremlin.
↪DriverRemoteConnection
>>> from goblin import Graph  # alias for aiogremlin.Graph

>>> async def go(loop):
...     remote_connection = await DriverRemoteConnection.open(
...         'http://localhost:8182/gremlin', 'g')
...     g = Graph().traversal().withRemote(remote_connection)
...     vertices = await g.V().toList()
...     await remote_connection.close()
...     return vertices

>>> results = loop.run_until_complete(go(loop))
>>> results
[v[...], ...]


>>> loop.run_until_complete(go(loop))
[v[...], ...]
```

For more information on using the `Graph`, see the aiogremlin documentation or the *GLV docs*

Driver

Submit scripts and bindings to the Gremlin Server:

```
>>> import asyncio
>>> from goblin import Cluster  # alias for aiogremlin.Cluster

>>> loop = asyncio.get_event_loop()
```

```
>>> async def go(loop):
...     cluster = await Cluster.open(loop)
...     client = await cluster.connect()
...     resp = await client.submit(
...         "g.addV('developer').property(k1, v1)",
...         bindings={'k1': 'name', 'v1': 'Leif'})
...     async for msg in resp:
...         print(msg)
...     await cluster.close()

>>> loop.run_until_complete(go(loop))
v[...]
```

For more information on using the driver, see the aiogremlin documentation or the *Driver docs*

Contents:

4.1.1 Using the OGM

Goblin aims to provide a powerful Object Graph Mapper **(OGM)** while maintaining a simple, transparent interface. This document describes the OGM components in more detail.

4.1.1.1 Modeling Graph Elements with `Goblin`

At the core of the *Goblin* is the concept of the graph element. TinkerPop 3 (TP3) uses three basic kinds of elements: Vertex, Edge, and Property. In order to achieve consistent mapping between Python objects and TP3 elements, *Goblin* provides three corresponding Python base classes that are used to model graph data: *Vertex*, *Edge*, and *Property*. While these classes are created to interact smoothly with TP3, it is important to remember that *Goblin* does not attempt to implement the same element interface found in TP3. Indeed, other than user defined properties, *Goblin* elements feature little to no interface. To begin modeling data, simply create *model* element classes that inherit from the *goblin.element* classes. For example:

```
import goblin
from gremlin_python import Cardinality


class Person(goblin.Vertex):
    pass


class City(goblin.Vertex):
    pass


class BornIn(goblin.Edge):
    pass
```

And that is it, these are valid element classes that can be saved to the graph database. Using these three classes we can model a series of people that are connected to the cities in which they were born. However, these elements aren't very useful, as they don't contain any information about the person or place. To remedy this, add some properties to the classes.

4.1.1.2 Using goblin.properties

Using the *properties* module is a bit more involved, but it is still pretty easy. It simply requires that you create properties that are defined as Python class attributes, and each property requires that you pass a *DataType* class **or** instance as the first positional argument. This data type, which is a concrete class that inherits from *DataType*, handles validation, as well as any necessary conversion when data is mapped between the database and the OGM. *Goblin* currently ships with 4 data types: *String*, *Integer*, *Float*, and *Boolean*. Example property definition:

```
>>> import goblin
>>> class Person(goblin.Vertex):
...     name = goblin.Property(goblin.String)
>>> class City(goblin.Vertex):
...      name = goblin.Property(goblin.String)
...      population = goblin.Property(goblin.Integer)
>>> class BornIn(goblin.Edge):
...      pass
```

Goblin properties can also be created with a default value, set by using the kwarg *default* in the class definition:

```
>>> class BornIn(goblin.Edge):
...     date = goblin.Property(goblin.String, default='unknown')
```

4.1.1.3 Creating Elements and Setting Property Values

Behind the scenes, a small metaclass (the only metaclass used in *Goblin*), substitutes a *PropertyDescriptor* for the *Property*, which provides a simple interface for defining and updating properties using Python's descriptor protocol:

```
>>> leif = Person()
>>> leif.name = 'Leif'

>>> detroit = City()
>>> detroit.name = 'Detroit'
>>> detroit.population =    5311449  # CSA population

# change a property value
>>> leif.name = 'Leifur'
```

In the case that an invalid property value is set, the validator will raise a *ValidationError* immediately:

```
>>> detroit.population = 'a lot of people'
Traceback (most recent call last):
  ...
goblin.exception.ValidationError: Not a valid integer: a lot of people
```

Creating Edges ———— Creating edges is very similar to creating vertices, except that edges require that a source (outV) and target (inV) vertex be specified. Both source and target nodes must be *Goblin vertices*. Furthermore, they must be created in the database before the edge. This is further discussed below in the *Session* section. Source and target vertices may be passed to the edge on instantiation, or added using the property interface:

```
>>> leif_born_in_detroit = BornIn(leif, detroit)
>>> # or
>>> leif_born_in_detroit = BornIn()
>>> leif_born_in_detroit.source = leif
>>> leif_born_in_detroit.target = detroit
```

```
>>> leif_born_in_detroit.date  # default value
'unknown'
```

4.1.1.4 Vertex Properties

In addition to the aforementioned elements, TP3 graphs also use a special kind of property, called a vertex property, that allows for list/set cardinality and meta-properties. To accommodate this, *Goblin* provides a class *VertexProperty* that can be used directly to create multi-cardinality properties:

```
>>> from gremlin_python.process.traversal import Cardinality
>>> class Person(goblin.Vertex):
...     name = goblin.Property(goblin.String)
...     nicknames = goblin.VertexProperty(
...         goblin.String, card=Cardinality.list_)


>>> david = Person()
>>> david.name = 'David'
>>> david.nicknames = ['Dave', 'davebshow']
```

Notice that the cardinality of the *VertexProperty* must be explicitly set using the *card* kwarg and the *Cardinality* enumerator.

class gremlin_python.process.traversal.**Cardinality**(**args*, ***kwds*)

list_ = 1

set_ = 2

single = 3

VertexProperty provides a different interface than the simple, key/value style *PropertyDescriptor* in order to accomodate more advanced functionality. For accessing multi-cardinality vertex properties, *Goblin* provides several helper classes called *managers*. The *managers* inherits from `list` or `set` (depending on the specified cardinality), and provide a simple API for accessing and appending vertex properties. To continue with the previous example, we see the *dave* element's nicknames:

```
>>> david.nicknames
[<VertexProperty(type=<...>, value=Dave), <VertexProperty(type=<...>,␣
↪value=davebshow)]
```

To add a nickname without replacing the earlier values, we simple *append* as if the manager were a Python `list`:

```
>>> david.nicknames.append('db')
>>> david.nicknames
[<VertexProperty(type=<...>, value=Dave), <VertexProperty(type=<...>,␣
↪value=davebshow), <VertexProperty(type=<...>, value=db)]
```

If this were a *VertexProperty* with a set cardinality, we would simply use *add* to achieve similar functionality.

Both *ListVertexPropertyManager* and *SetVertexPropertyManager* provide a simple way to access a specific *VertexProperty*. You simply call the manager, passing the value of the vertex property to be accessed:

```
>>> db = david.nicknames('db')
```

The value of the vertex property can be accessed using the *value* property:

```
>>> db.value
'db'
```

4.1.1.5 Meta-properties

VertexProperty can also be used as a base classes for user defined vertex properties that contain meta-properties. To create meta-properties, define a custom vertex property class just like you would any other element, adding as many simple (non-vertex) properties as needed:

```
>>> class HistoricalName(goblin.VertexProperty):
...     notes = goblin.Property(goblin.String)
```

Now, the custom *VertexProperty* can be added to a vertex class, using any cardinality:

```
>>> class City(goblin.Vertex):
...     name = goblin.Property(goblin.String)
...     population = goblin.Property(goblin.Integer)
...     historical_name = HistoricalName(
...         goblin.String, card=Cardinality.list_)
```

Now, meta-properties can be set on the *VertexProperty* using the descriptor protocol:

```
>>> montreal = City()
>>> montreal.historical_name = ['Ville-Marie']
>>> montreal.historical_name('Ville-Marie').notes = 'Changed in 1705'
```

And that's it.

4.1.1.6 Saving Elements to the Database Using `Session`

All interaction with the database is achieved using the *Session* object. A *Goblin* session should not be confused with a Gremlin Server session, although in future releases it will provide support for server sessions and transactions. Instead, the *Session* object is used to save elements and spawn Gremlin traversals. Furthemore, any element created using a session is *live* in the sense that a *Session* object maintains a reference to session elements, and if a traversal executed using a session returns different property values for a session element, these values are automatically updated on the session element. Note - the examples shown in this section must be wrapped in coroutines and ran using the `asyncio.BaseEventLoop`, but, for convenience, they are shown as if they were run in a Python interpreter. To use a *Session*, first create a *Goblin App* using *Goblin.open*,

```
app = await goblin.Goblin.open(loop)
```

then register the defined element classes:

```
>>> app.register(Person, City, BornIn)
```

Goblin application support a variety of configuration options, for more information see *the Goblin application documentation*.

The best way to create elements is by adding them to the session, and then flushing the *pending* queue, thereby creating the elements in the database. The order in which elements are added **is** important, as elements will be created based on the order in which they are added. Therefore, when creating edges, it is important to add the source and target nodes before the edge (if they don't already exits). Using the previously created elements:

```
>>> async def create_elements(app):
...     session = await app.session()
...     session.add(leif, detroit, leif_born_in_detroit)
...     await session.flush()
>>> loop.run_until_complete(create_elements(app))
```

And that is it. To see that these elements have actually been created in the db, check that they now have unique ids assigned to them:

```
>>> assert leif.id
>>> assert detroit.id
>>> assert leif_born_in_detroit.id
```

For more information on the *Goblin App*, please see *Using the Goblin App*

Session provides a variety of other CRUD functions, but all creation and updating can be achieved simply using the *add* and *flush* methods.

4.1.1.7 Writing Custom Gremlin Traversals

Finally, *Session* objects allow you to write custom Gremlin traversals using the official gremlin-python Gremlin Language Variant **(GLV)**. There are two methods available for writing session based traversals. The first, *traversal*, accepts an element class as a positional argument. This is merely for convenience, and generates this equivalent Gremlin:

```
>>> session = loop.run_until_complete(app.session())
>>> session.traversal(Person)
[['V'], ['hasLabel', 'person']]
```

Or, simply use the property *g*:

```
>>> session.g.V().hasLabel('person')
[['V'], ['hasLabel', 'person']]
```

In general property names are mapped directly from the OGM to the database. However, by passing the *db_name* kwarg to a property definition, the user has the ability to override this behavior. To avoid mistakes in the case of custom database property names, it is encouraged to access the mapped property names as class attributes:

```
>>> Person.name
'name'
```

So, to write a traversal:

```
>>> session.traversal(Person).has(Person.name, 'Leifur')
[['V'], ['hasLabel', 'person'], ['has', 'name', 'Leifur']]
```

Also, it is important to note that certain data types could be transformed before they are written to the database. Therefore, the data type method *to_db* may be required:

```
>>> session.traversal(Person).has(
...     Person.name, goblin.String().to_db('Leifur'))
[['V'], ['hasLabel', 'person'], ['has', 'name', 'Leifur']]
```

While this is not the case with any of the simple data types shipped with *Goblin*, custom data types or future additions may require this kind of operation. Because of this, *Goblin* includes the convenience function *bindprop*, which also allows an optional binding for the value to be specified:

```
>>> from goblin.session import bindprop
>>> traversal = session.traversal(Person)
>>> traversal.has(bindprop(Person, 'name', 'Leifur', binding='v1'))
[['V'], ['hasLabel', 'person'], ['has', binding[name=binding[v1=Leifur]]]]
```

Finally, there are a variety of ways to to submit a traversal to the server. First of all, all traversals are themselve asynchronous iterators, and using them as such will cause a traversal to be sent on the wire:

```
async for msg in session.g.V().hasLabel('person'):
    print(msg)
```

Furthermore, *Goblin* provides several convenience methods that submit a traversal as well as process the results `toList`, `toSet` and `next`. These methods both submit a script to the server and iterate over the results. Remember to *await* the traversal when calling these methods:

```
traversal = session.traversal(Person)
leif = await traversal.has(
    bindprop(Person, 'name', 'Leifur', binding='v1')).next()
```

And that is pretty much it. We hope you enjoy the *Goblin* OGM.

4.1.2 Using Graph (GLV)

Goblin provides access to the underlying `aiogremlin` asynchronous version of the Gremlin-Python Gremlin Language Variant (GLV) that is bundled with Apache TinkerPop beginning with the 3.2.2 release. Traversals are generated using the class `Graph` combined with a remote connection class, either `DriverRemoteConnection`:

```
>>> import asyncio
>>> import goblin  # provides aliases to common aiogremlin objects

>>> loop = asyncio.get_event_loop()
>>> remote_conn = loop.run_until_complete(
...     goblin.DriverRemoteConnection.open(
...         "http://localhost:8182/gremlin", 'g'))
>>> graph = goblin.driver.Graph()
>>> g = graph.traversal().withRemote(remote_conn)
```

Once you have a traversal source, it's all Gremlin...:

```
>>> traversal = g.addV('query_language').property('name', 'gremlin')
```

traversal is in an instance of `AsyncGraphTraversal`, which implements the Python 3.5 asynchronous iterator protocol:

```
>>> async def iterate_traversal(traversal):
...     async for msg in traversal:
...         print(msg)

>>> loop.run_until_complete(iterate_traversal(traversal))
v[...]
```

`AsyncGraphTraversal` also provides several convenience coroutine methods to help iterate over results:

- `next`
- `toList`

- `toSet`

Notice the mixedCase? Not very pythonic? Well no, but it maintains continuity with the Gremlin query language, and that's what the GLV is all about...

Note: Gremlin steps that are reserved words in Python, like *or*, *in*, use a a trailing underscore *or_* and *in_*.

4.1.2.1 The Side Effect Interface

When using TinkerPop 3.2.2+ with the default *Goblin* provides an asynchronous side effects interface using the `AsyncRemoteTraversalSideEffects` class. This allows side effects to be retrieved after executing the traversal:

```
>>> traversal = g.V().aggregate('a')
>>> loop.run_until_complete(traversal.iterate())
[['V'], ['aggregate', 'a']]
```

Calling `keys` will return an asynchronous iterator containing all keys for cached side effects:

```
>>> async def get_side_effect_keys(traversal):
...     keys = await traversal.side_effects.keys()
...     print(keys)
```

```
>>> loop.run_until_complete(get_side_effect_keys(traversal))
{'a'}
```

Then calling `get` using a valid key will return the cached side effects:

```
>>> async def get_side_effects(traversal):
...     se = await traversal.side_effects.get('a')
...     print(se)


>>> loop.run_until_complete(get_side_effects(traversal))
{v[1]: 1, ...}
```

And that's it! For more information on Gremlin Language Variants, please visit the Apache TinkerPop GLV Documentation.

4.1.3 Using the Driver

4.1.3.1 Connecting to a `Cluster`

To take advantage of the higher level features of the *driver*, *Goblin* provides the `Cluster` object. `Cluster` is used to create multi-host clients that leverage connection pooling and sharing. Its interface is based on the TinkerPop Java driver:

```
>>> async def print_results(gremlin='1+1'):
...     # opens a cluster with default config
...     cluster = await driver.Cluster.open('')
...     client = await cluster.connect()
...     # round robin requests to available hosts
...     resp = await client.submit(gremlin=gremlin)
...     async for msg in resp:
...         print(msg)
...     await cluster.close()  # Close all connections to all hosts
```

And that is it. While `Cluster` is simple to learn and use, it provides a wide variety of configuration options.

4.1.3.2 Configuring `Cluster`

Configuration options can be set on `Cluster` in one of two ways, either passed as keyword arguments to `open`, or stored in a configuration file and passed to the `open` using the kwarg *configfile*. Configuration files can be either YAML or JSON format. Currently, `Cluster` uses the following configuration:

Key	Description	Default
scheme	URI scheme, typically 'ws' or 'wss' for secure websockets	'ws'
hosts	A list of hosts the cluster will connect to	['localhost']
port	The port of the Gremlin Server to connect to, same for all hosts	8182
ssl_certfile	File containing ssl certificate	''
ssl_keyfile	File containing ssl key	''
ssl_password	File containing password for ssl keyfile	''
username	Username for Gremlin Server authentication	''
password	Password for Gremlin Server authentication	''
response_timeout	Timeout for reading responses from the stream	*None*
max_conns	The maximum number of connections open at any time to this host	4
min_conns	The minimum number of connection open at any time to this host	1
max_times_acquired	The maximum number of times a single pool connection can be acquired and shared	16
max_inflight	The maximum number of unresolved messages that may be pending on any one connection	64
message_serializer	String denoting the class used for message serialization, currently only supports basic GraphSONMessageSerializer	'classpath'

For information related to improving driver performance, please refer to the *performance section*.

4.1.4 Configuring the `Goblin` App Object

The *`Goblin`* object generally supports the same configuration options as `Cluster`. Please see the *driver docs* for a complete list of configuration parameters.

The *`Goblin`* object should be created using the *`open`* classmethod, and configuration can be passed as keyword arguments, or loaded from a config file:

```
>>> import asyncio
>>> from goblin import Goblin

>>> loop = asyncio.get_event_loop()

>>> app = loop.run_until_complete(Goblin.open(loop))
>>> app.config_from_file('config.yml')
```

Contents of *config.yml*:

```
scheme: 'ws'
hosts: ['localhost']
port': 8182
```

```
ssl_certfile: ''
ssl_keyfile: ''
ssl_password: ''
username: ''
password: ''
response_timeout: null
max_conns: 4
min_conns: 1
max_times_acquired: 16
max_inflight: 64
message_serializer: 'goblin.driver.GraphSONMessageSerializer'
```

4.1.4.1 Special `Goblin` App Configuration

Goblin supports two additional configuration keyword parameters: *aliases* and *get_hashable_id*.

aliases

aliases as stated in the TinkerPop docs: are "a map of key/value pairs that allow globally bound Graph and TraversalSource objects to be aliased to different variable names for purposes of the current request". Setting the aliases on the *Goblin* object provides a default for this value to be passed on each request.

get_hashable_id

get_hashable_id is a callable that translates a graph id into a hash that can be used to map graph elements in the *Session* element cache. In many cases, it is not necessary to provide a value for this keyword argument. For example, TinkerGraph assigns integer IDs that work perfectly for this purpose. However, other provider implementations, such as DSE, use more complex data structures to represent element IDs. In this case, the application developer must provide a hashing function. For example, the following recipe takes an id map and uses its values to produces a hashable id:

```
>>> def get_id_hash(dict):
...     hashes = map(hash, dict.items())
...     id_hash = functools.reduce(operator.xor, hashes, 0)
...     return id_hash
```

Look for provider specific *Goblin* libraries in the near future!

4.1.5 Improving Driver Performance

The *goblin.driver* aims to be as performant as possible, yet it is potentially limited by implementation details, as well as its underlying software stack i.e., the websocket client, the event loop implementation, etc. If necessary, a few tricks can boost its performance.

4.1.5.1 Use `cython`

Before installing *Goblin*, install `cython`:

```
$ pip install cython
```

4.1.5.2 Use `ujson`

Install `ujson` to speed up serialzation:

```
$ pip install ujson
```

4.1.5.3 Use `uvloop`

Install `uvloop`, a Cython implementation of an event loop:

```
$ pip install uvloop
```

Then, in application code, set the `asyncio.set_event_loop_policy()`:

```
>>> import asyncio
>>> import uvloop
>>> asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
```

4.1.6 Goblin API

4.1.6.1 goblin package

Subpackages

goblin.driver package

Contains aliases to classes from *aiogremlin*:

Submodules

goblin.abc module

class goblin.abc.**BaseProperty**
Bases: `object`

Abstract base class that implements the property interface

data_type

class goblin.abc.**DataType**(*val=None*)
Bases: `abc.ABC`

Abstract base class for Goblin Data Types. All custom data types should inherit from *DataType*.

to_db(*val=None*)
Convert property value to db compatible format. If no value passed, try to use default bound value

to_ogm(*val*)
Convert property value to a Python compatible format

validate(*val*)
Validate property value

validate_vertex_prop(*val*, *card*, *vertex_prop*, *data_type*)

goblin.app module

Goblin application class and class constructor

class goblin.app.**Goblin**(*cluster*, *, *provider=<class 'goblin.provider.TinkerGraph'>*, *get_hashable_id=None*, *aliases=None*)

Bases: object

Class used to encapsulate database connection configuration and generate database connections Used as a factory to create *Session* objects.

Parameters

- **url** (*str*) – Database url
- **loop** (*asyncio.BaseEventLoop*) – Event loop implementation
- **features** (*dict*) – Vendor implementation specific database features
- **config** (*dict*) – Config parameters for application

close()

cluster

config

config_from_file(*filename*)
Load configuration from from file.

Parameters **filename** (*str*) – Path to the configuration file.

config_from_json(*filename*)
Load configuration from from JSON file.

Parameters **filename** (*str*) – Path to the configuration file.

config_from_module(*module*)

config_from_yaml(*filename*)

edges
Registered edge classes

classmethod open(*loop*, *, *provider=<class 'goblin.provider.TinkerGraph'>*, *get_hashable_id=None*, *aliases=None*, ***config*)

register(**elements*)
Register user created Element classes.

Parameters **elements** (goblin.element.Element) – User defined Element classes

register_from_module(*module*, *, *package=None*)

session(*, *processor=''*, *op='eval'*, *aliases=None*)
Create a session object.

Returns *Session* object

url
Database url

vertices
Registered vertex classes

goblin.element module

Module defining graph elements.

class goblin.element.**Edge**(*source=None*, *target=None*)
Bases: *goblin.element.Element*

Base class for user defined Edge classes.

Parameters

- **source** (Vertex) – Source (outV) vertex
- **target** (Vertex) – Target (inV) vertex

delsource()

deltarget()

classmethod from_dict(*d*)

getsource()

gettarget()

setsource(*vertex*)

settarget(*vertex*)

source

target

to_dict(*source=None*, *target=None*)

class goblin.element.**Element**(***kwargs*)
Bases: object

Base class for classes that implement the Element property interface

id

class goblin.element.**ElementMeta**
Bases: type

Metaclass for graph elements. Responsible for creating the *Mapping* object and replacing user defined *goblin.properties.Property* with *goblin.properties.PropertyDescriptor*.

class goblin.element.**GenericEdge**(*source=None*, *target=None*)
Bases: *goblin.element.Edge*

Class used to build edges when user defined edges class is not available. Generally not instantiated by end user.

class goblin.element.**GenericVertex**(***kwargs*)
Bases: *goblin.element.Vertex*

Class used to build vertices when user defined vertex class is not available. Generally not instantiated by end user.

class goblin.element.**Vertex**(***kwargs*)
Bases: *goblin.element.Element*

Base class for user defined Vertex classes

classmethod from_dict(*d*)

to_dict()

class goblin.element.**VertexProperty**(*data_type*, *, *default=None*, *db_name=None*, *card=None*, *db_name_factory=None*)

Bases: *goblin.element.Element*, *goblin.abc.BaseProperty*

Base class for user defined vertex properties.

cardinality

data_type

db_name

db_name_factory

default

from_dict(*d*)

getdb_name()

getvalue()

setgetdb_name(*val*)

setvalue(*val*)

to_dict()

value

class goblin.element.**VertexPropertyDescriptor**(*name*, *vertex_property*)

Bases: object

Descriptor that validates user property input and gets/sets properties as instance attributes.

goblin.exception module

exception goblin.exception.**ClientError**

Bases: Exception

exception goblin.exception.**ConfigError**

Bases: Exception

exception goblin.exception.**ConfigurationError**

Bases: Exception

exception goblin.exception.**ElementError**

Bases: Exception

exception goblin.exception.**GremlinServerError**

Bases: Exception

exception goblin.exception.**MappingError**

Bases: Exception

exception goblin.exception.**ResponseTimeoutError**

Bases: Exception

exception goblin.exception.**ValidationError**

Bases: Exception

goblin.manager module

Managers for multi cardinality vertex properties

class goblin.manager.**ListVertexPropertyManager**(*data_type*, *vertex_prop*, *card*, *obj*)
Bases: list, *goblin.manager.VertexPropertyManager*

append(*val*)

vp_map

class goblin.manager.**SetVertexPropertyManager**(*data_type*, *vertex_prop*, *card*, *obj*)
Bases: set, *goblin.manager.VertexPropertyManager*

add(*val*)

class goblin.manager.**VertexPropertyManager**(*data_type*, *vertex_prop*, *card*)
Bases: object

mapper_func

goblin.mapper module

Helper functions and class to map between OGM Elements <-> DB Elements

class goblin.mapper.**Mapping**(*namespace*, *element_type*, *mapper_func*, *properties*)
Bases: object

This class stores the information necessary to map between an OGM element and a DB element.

db_properties
A dictionary of property mappings

label
Element label

mapper_func
Function responsible for mapping db results to ogm

ogm_properties
A dictionary of property mappings

goblin.mapper.**create_mapping**(*namespace*, *properties*)
Constructor for *Mapping*

goblin.mapper.**get_hashable_id**(*val*)

goblin.mapper.**get_metaprops**(*vertex_property*, *mapping*)

goblin.mapper.**map_edge_to_ogm**(*result*, *props*, *element*, *, *mapping=None*)
Map an edge returned by DB to OGM edge

goblin.mapper.**map_props_to_db**(*element*, *mapping*)
Convert OGM property names/values to DB property names/values

goblin.mapper.**map_vertex_property_to_ogm**(*result*, *element*, *, *mapping=None*)
Map a vertex returned by DB to OGM vertex

goblin.mapper.**map_vertex_to_ogm**(*result*, *props*, *element*, *, *mapping=None*)
Map a vertex returned by DB to OGM vertex

goblin.properties module

Classes to handle properties and data type definitions

class goblin.properties.**Boolean**(*val=None*)
Bases: *goblin.abc.DataType*

Simple boolean datatype

to_db(*val=None*)

to_ogm(*val*)

validate(*val*)

class goblin.properties.**Float**(*val=None*)
Bases: *goblin.abc.DataType*

Simple float datatype

to_db(*val=None*)

to_ogm(*val*)

validate(*val*)

class goblin.properties.**Generic**(*val=None*)
Bases: *goblin.abc.DataType*

to_db(*val=None*)

to_ogm(*val*)

validate(*val*)

class goblin.properties.**IdProperty**(*data_type*, *, *serializer=None*)
Bases: *goblin.abc.BaseProperty*

data_type

serializer

class goblin.properties.**IdPropertyDescriptor**(*name*, *prop*)
Bases: object

class goblin.properties.**Integer**(*val=None*)
Bases: *goblin.abc.DataType*

Simple integer datatype

to_db(*val=None*)

to_ogm(*val*)

validate(*val*)

class goblin.properties.**Property**(*data_type*, *, *db_name=None*, *default=None*, *db_name_factory=None*)
Bases: *goblin.abc.BaseProperty*

API class used to define properties. Replaced with *PropertyDescriptor* by *goblin.element.ElementMeta*.

Parameters

- **data_type** (goblin.abc.DataType) – Str or class of data type
- **db_name** (*str*) – User defined custom name for property in db

- **default** – Default value for this property.

data_type

db_name

db_name_factory

default

getdb_name()

setgetdb_name(*val*)

class goblin.properties.**PropertyDescriptor**(*name*, *prop*)
Bases: object

Descriptor that validates user property input and gets/sets properties as instance attributes. Not instantiated by user.

class goblin.properties.**String**(*val=None*)
Bases: *goblin.abc.DataType*

Simple string datatype

to_db(*val=None*)

to_ogm(*val*)

validate(*val*)

goblin.properties.**default_id_serializer**(*val*)

goblin.properties.**noop_factory**(*x*, *y*)

goblin.session module

Main OGM API classes and constructors

class goblin.session.**Session**(*app*, *remote_connection*, *get_hashable_id*)
Bases: object

Provides the main API for interacting with the database. Does not necessarily correpsond to a database session. Don't instantiate directly, instead use *Goblin.session*.

Parameters

- **app** (goblin.app.Goblin) –
- **conn** (*aiogremlin.driver.connection.Connection*) –

add(**elements*)
Add elements to session pending queue.

Parameters **elements** (goblin.element.Element) – Elements to be added

app

close()

current

flush()
Issue creation/update queries to database for all elements in the session pending queue.

g
Get a simple traversal source.

Returns *gremlin_python.process.GraphTraversalSource* object

get_edge(*edge*)
Get a edge from the db. Edge must have id.

Parameters **element** (goblin.element.Edge) – Edge to be retrieved

Returns *Edge* | None

get_vertex(*vertex*)
Get a vertex from the db. Vertex must have id.

Parameters **element** (goblin.element.Vertex) – Vertex to be retrieved

Returns *Vertex* | None

graph

remote_connection

remove_edge(*edge*)
Remove an edge from the db.

Parameters **edge** (goblin.element.Edge) – Element to be removed

remove_vertex(*vertex*)
Remove a vertex from the db.

Parameters **vertex** (goblin.element.Vertex) – Vertex to be removed

save(*elem*)
Save an element to the db.

Parameters **element** (goblin.element.Element) – Vertex or Edge to be saved

Returns *Element* object

save_edge(*edge*)
Save an edge to the db.

Parameters **element** (goblin.element.Edge) – Edge to be saved

Returns *Edge* object

save_vertex(*vertex*)
Save a vertex to the db.

Parameters **element** (goblin.element.Vertex) – Vertex to be saved

Returns *Vertex* object

submit(*bytecode*)
Submit a query to the Gremiln Server.

Parameters

- **gremlin** (*str*) – Gremlin script to submit to server.
- **bindings** (*dict*) – A mapping of bindings for Gremlin script.

Returns *gremlin_python.driver.remove_connection.RemoteTraversal* object

traversal(*element_class=None*)
Generate a traversal using a user defined element class as a starting point.

Parameters **element_class** (goblin.element.Element) – An optional element class that will dictate the element type (vertex/edge) as well as the label for the traversal source

Returns *aiogremlin.process.graph_traversal.AsyncGraphTraversal*

goblin.session.**bindprop**(*element_class*, *ogm_name*, *val*, *, *binding=None*)

Helper function for binding ogm properties/values to corresponding db properties/values for traversals.

Parameters

- **element_class** (goblin.element.Element) – User defined element class
- **ogm_name** (*str*) – Name of property as defined in the ogm
- **val** – The property value
- **binding** (*str*) – The binding for val (optional)

Returns tuple object ('db_property_name', ('binding(if passed)', val))

Module contents

Python toolkit for Tinker Pop 3 Gremlin Server.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

goblin, 28
goblin.abc, 20
goblin.app, 21
goblin.driver, 20
goblin.element, 22
goblin.exception, 23
goblin.manager, 24
goblin.mapper, 24
goblin.properties, 25
goblin.session, 26

Index

A

add() (goblin.manager.SetVertexPropertyManager method), 24
add() (goblin.session.Session method), 26
app (goblin.session.Session attribute), 26
append() (goblin.manager.ListVertexPropertyManager method), 24

B

BaseProperty (class in goblin.abc), 20
bindprop() (in module goblin.session), 28
Boolean (class in goblin.properties), 25

C

Cardinality (class in gremlin_python.process.traversal), 13
cardinality (goblin.element.VertexProperty attribute), 23
ClientError, 23
close() (goblin.app.Goblin method), 21
close() (goblin.session.Session method), 26
cluster (goblin.app.Goblin attribute), 21
config (goblin.app.Goblin attribute), 21
config_from_file() (goblin.app.Goblin method), 21
config_from_json() (goblin.app.Goblin method), 21
config_from_module() (goblin.app.Goblin method), 21
config_from_yaml() (goblin.app.Goblin method), 21
ConfigError, 23
ConfigurationError, 23
create_mapping() (in module goblin.mapper), 24
current (goblin.session.Session attribute), 26

D

data_type (goblin.abc.BaseProperty attribute), 20
data_type (goblin.element.VertexProperty attribute), 23
data_type (goblin.properties.IdProperty attribute), 25
data_type (goblin.properties.Property attribute), 26
DataType (class in goblin.abc), 20
db_name (goblin.element.VertexProperty attribute), 23
db_name (goblin.properties.Property attribute), 26
db_name_factory (goblin.element.VertexProperty attribute), 23
db_name_factory (goblin.properties.Property attribute), 26
db_properties (goblin.mapper.Mapping attribute), 24
default (goblin.element.VertexProperty attribute), 23
default (goblin.properties.Property attribute), 26
default_id_serializer() (in module goblin.properties), 26
delsource() (goblin.element.Edge method), 22
deltarget() (goblin.element.Edge method), 22

E

Edge (class in goblin.element), 22
edges (goblin.app.Goblin attribute), 21
Element (class in goblin.element), 22
ElementError, 23
ElementMeta (class in goblin.element), 22

F

Float (class in goblin.properties), 25
flush() (goblin.session.Session method), 26
from_dict() (goblin.element.Edge class method), 22
from_dict() (goblin.element.Vertex class method), 22
from_dict() (goblin.element.VertexProperty method), 23

G

g (goblin.session.Session attribute), 26
Generic (class in goblin.properties), 25
GenericEdge (class in goblin.element), 22
GenericVertex (class in goblin.element), 22
get_edge() (goblin.session.Session method), 27
get_hashable_id() (in module goblin.mapper), 24
get_metaprops() (in module goblin.mapper), 24
get_vertex() (goblin.session.Session method), 27
getdb_name() (goblin.element.VertexProperty method), 23
getdb_name() (goblin.properties.Property method), 26
getsource() (goblin.element.Edge method), 22
gettarget() (goblin.element.Edge method), 22

getvalue() (goblin.element.VertexProperty method), 23
Goblin (class in goblin.app), 21
goblin (module), 28
goblin.abc (module), 20
goblin.app (module), 21
goblin.driver (module), 20
goblin.element (module), 22
goblin.exception (module), 23
goblin.manager (module), 24
goblin.mapper (module), 24
goblin.properties (module), 25
goblin.session (module), 26
graph (goblin.session.Session attribute), 27
GremlinServerError, 23

I

id (goblin.element.Element attribute), 22
IdProperty (class in goblin.properties), 25
IdPropertyDescriptor (class in goblin.properties), 25
Integer (class in goblin.properties), 25

L

label (goblin.mapper.Mapping attribute), 24
list_ (gremlin_python.process.traversal.Cardinality attribute), 13
ListVertexPropertyManager (class in goblin.manager), 24

M

map_edge_to_ogm() (in module goblin.mapper), 24
map_props_to_db() (in module goblin.mapper), 24
map_vertex_property_to_ogm() (in module goblin.mapper), 24
map_vertex_to_ogm() (in module goblin.mapper), 24
mapper_func (goblin.manager.VertexPropertyManager attribute), 24
mapper_func (goblin.mapper.Mapping attribute), 24
Mapping (class in goblin.mapper), 24
MappingError, 23

N

noop_factory() (in module goblin.properties), 26

O

ogm_properties (goblin.mapper.Mapping attribute), 24
open() (goblin.app.Goblin class method), 21

P

Property (class in goblin.properties), 25
PropertyDescriptor (class in goblin.properties), 26

R

register() (goblin.app.Goblin method), 21
register_from_module() (goblin.app.Goblin method), 21
remote_connection (goblin.session.Session attribute), 27
remove_edge() (goblin.session.Session method), 27
remove_vertex() (goblin.session.Session method), 27
ResponseTimeoutError, 23

S

save() (goblin.session.Session method), 27
save_edge() (goblin.session.Session method), 27
save_vertex() (goblin.session.Session method), 27
serializer (goblin.properties.IdProperty attribute), 25
Session (class in goblin.session), 26
session() (goblin.app.Goblin method), 21
set_ (gremlin_python.process.traversal.Cardinality attribute), 13
setgetdb_name() (goblin.element.VertexProperty method), 23
setgetdb_name() (goblin.properties.Property method), 26
setsource() (goblin.element.Edge method), 22
settarget() (goblin.element.Edge method), 22
setvalue() (goblin.element.VertexProperty method), 23
SetVertexPropertyManager (class in goblin.manager), 24
single (gremlin_python.process.traversal.Cardinality attribute), 13
source (goblin.element.Edge attribute), 22
String (class in goblin.properties), 26
submit() (goblin.session.Session method), 27

T

target (goblin.element.Edge attribute), 22
to_db() (goblin.abc.DataType method), 20
to_db() (goblin.properties.Boolean method), 25
to_db() (goblin.properties.Float method), 25
to_db() (goblin.properties.Generic method), 25
to_db() (goblin.properties.Integer method), 25
to_db() (goblin.properties.String method), 26
to_dict() (goblin.element.Edge method), 22
to_dict() (goblin.element.Vertex method), 22
to_dict() (goblin.element.VertexProperty method), 23
to_ogm() (goblin.abc.DataType method), 20
to_ogm() (goblin.properties.Boolean method), 25
to_ogm() (goblin.properties.Float method), 25
to_ogm() (goblin.properties.Generic method), 25
to_ogm() (goblin.properties.Integer method), 25
to_ogm() (goblin.properties.String method), 26
traversal() (goblin.session.Session method), 27

U

url (goblin.app.Goblin attribute), 21

V

validate() (goblin.abc.DataType method), 20
validate() (goblin.properties.Boolean method), 25
validate() (goblin.properties.Float method), 25

validate() (goblin.properties.Generic method), 25
validate() (goblin.properties.Integer method), 25
validate() (goblin.properties.String method), 26
validate_vertex_prop() (goblin.abc.DataType method), 20
ValidationError, 23
value (goblin.element.VertexProperty attribute), 23
Vertex (class in goblin.element), 22
VertexProperty (class in goblin.element), 22
VertexPropertyDescriptor (class in goblin.element), 23
VertexPropertyManager (class in goblin.manager), 24
vertices (goblin.app.Goblin attribute), 21
vp_map (goblin.manager.ListVertexPropertyManager attribute), 24