aiocouch

Release 3.0.0

Mario Bielert

Aug 25, 2023

CONTENTS

Asynchronous CouchDB client library for asyncio and Python.

Current version is 3.0.0.

CHAPTER

ONE

KEY FEATURES

- All requests are asynchronus using aiohttp
- Supports CouchDB 2.x and 3.x
- Support for modern Python 3.7

CHAPTER

TWO

LIBRARY INSTALLATION

```
pip install aiocouch
```

CHAPTER

THREE

GETTING STARTED

The following code retrieves and prints the list of `incredients` of the `apple_pie` `recipe`. The `incredients` are stored as a list in the `apple_pie` *Document*, which is part of the `recipe` *Database*. We use the context manager *CouchDB* to create a new session.

```
from aiocouch import CouchDB

async with CouchDB(
    "http://localhost:5984", user="admin", password="admin"
) as couchdb:
    db = await couchdb["recipes"]
    doc = await db["apple_pie"]
    print(doc["incredients"])
```

We can also create new recipes, for instance for some delicious cookies.

```
    new_doc = await db.create(
        "cookies", data={"title": "Granny's cookies", "rating": ""}
    )
    await new_doc.save()
#
```

CHAPTER

FOUR

TUTORIAL

You can find a more in-depth discussion of the core concepts and next steps at the *Introduction* page.

CHAPTER

FIVE

SOURCE CODE

The project is hosted on GitHub.

Please feel free to file an issue on the bug tracker if you have found a bug or have some suggestion in order to improve the library.

The library uses GitHub Actions for Continuous Integration.

CHAPTER

SIX

DEPENDENCIES

- Python 3.7+
- *aiohttp*
- *Deprecated*

CHAPTER

SEVEN

AUTHORS AND LICENSE

The `aiocouch` package is written mostly by Mario Bielert.

It's *BSD 3-clause* licensed and freely available.

Feel free to improve this package and send a pull request to GitHub.

CHAPTER

EIGHT

TABLE OF CONTENTS

8.1 Introduction

This short tutorial will give you an overview of the *aiocouch* library.

You can directly jump to particular sections using the *Navigation* on the left.

8.2 Session

Every request to the CouchDB server is embedded into a session. A session is represented by an instance of *aiocouch.CouchDB*. A session can be created using the constructor of the class or by using the class as a context manager.

8.2.1 Examples

Create a session with the context manager

```
async with aiocouch.CouchDB("http://localhost") as couchdb:
    await couchdb.check_credentials()
```

Note that the session will be closed, once the scope of the with statement is left.

A session can also be handled using variables. The session needs to be closed manually.

```
couchdb = aiocouch.CouchDB("http://localhost")
await couchdb.check_credentials()
await couchdb.close()
```

8.2.2 Reference

class aiocouch.**CouchDB**(**args*, ***kwargs*)

CouchDB Server Connection Session

The

Parameters

- **server** (*str*) – URL of the CouchDB server
- **user** (*str*) – user used for authentication
- **password** (*str*) – password for authentication

- **cookie** (*str*) – The session cookie used for authentication
- **kwargs** (*Any*) – Any other kwargs are passed to `aiohttp.ClientSession`

`await __getitem__(id)`

Returns a representation for the given database identifier

Raises
NotFoundError – if the database does not exist

Parameters
id (*str*) – The identifier of the database

Return type
Database

Returns
The representation of the database

`await check_credentials()`

Check the provided credentials.

Raises
UnauthorizedError – if provided credentials aren't valid

Return type
None

`await close()`

Closes the connection to the CouchDB server

Return type
None

`await create(id, exists_ok=False, **kwargs)`

Creates a new database on the server

Raises
PreconditionFailedError – if the database already exists and `exists_ok` is `False`

Parameters

- **id** (*str*) – the identifier of the database
- **exists_ok** (*bool*) – If `True`, don't raise if the database exists

Return type
Database

Returns
Returns a representation for the created database

`await info()`

Returns the meta information about the connected CouchDB server.

See also `GET /`.

Return type
`Dict[str, Any]`

Returns
A dict containing the response json.

await keys(***params*)

Returns all database names

Return type
List[str]

Returns
A list containing the names of all databases on the server

8.3 Databases

Once you have established a session with the server, you need a *Database* instance to access the data. A Database instance is an representation of a database on the server. Database instances allow to access *Document* instances. Also, Database instances can be used to configure the user and group permissions.

8.3.1 Getting a Database instance

While the constructor of the *Database* class can be used to get a representation of a specific database, the canonical way to get an instance are the member functions of the *CouchDB* class.

The following code returns an instance for the *animals* database.

```
animals = await session["animals"]
```

aiocouch only allows to get an instance for a database that exists on the server.

8.3.2 Creating new databases

To create a new database on the server, the *create()* method of the session object is used.

```
animals = await session.create("animals")
```

By default, *aiocouch* only allows to use the create method for a database that does not exist on the server.

8.3.3 Listing documents

The _all_docs view allows to retrieve all documents stored in a given database on the server. *aiocouch* also exposes this view as methods of the database class.

The method *docs()* allows to retrieve documents by a list of ids or all documents with ids matching a given prefix. Similar to a dict, all documents of a database can be iterated with the methods *akeys()*, and *values()*.

To perform more sophisticated document selections, the method *find()* allows to search for documents matching the complex selector syntax of CouchDB.

8.3.4 Reference

class aiocouch.database.**Database**(*couchdb*, *id*)

A local representation for the referenced CouchDB database

An instance of this class represents a local copy of a CouchDB database. It allows to create and retrieve *Document* instances, as well as the iteration other many documents.

Variables
id – the id of the database

Parameters

- **couchdb** (CouchDB) – The CouchDB connection session
- **id** (str) – the id of the database

await __getitem__(*id*)

Returns the document with the given id

Raises
NotFoundError – if the given document does not exist

Parameters
id (str) – the name of the document

Return type
Document

Returns
a local copy of the document

async for ... in akeys(***params*)

A generator returning the names of all documents in the database

Parameters
params (Any) – passed into aiohttp.ClientSession.request()

Return type
AsyncGenerator[str, None]

Returns
returns all document ids

property all_docs: AllDocsView

Returns the all_docs view of the database

Returns
Description of returned object.

async for ... in changes(*last_event_id=None*, ***params*)

Listens for events made to documents of this database

This will return *DeletedEvent* and *ChangedEvent* for deleted and modified documents, respectively.

See also /db/_changes.

For convenience, the `last-event-id` parameter can also be passed as `last_event_id`.

Return type
AsyncGenerator[*BaseChangeEvent*, None]

await create(*id*, *exists_ok=False*, *data=None*)

Returns a local representation of a new document in the database

This method will check if a document with the given name already exists and return a *Document* instance.

This method does not create a document with the given name on the server. You need to call *save()* on the returned document.

Raises
ConflictError – if the document does not exist on the server

Parameters

- **id** (str) – the name of the document
- **exists_ok** (bool) – If *True*, do not raise a *ConflictError* if an document with the given name already exists. Instead return the existing document.
- **data** (Optional[Dict[str, Any]]) – Description of parameter *data*. Defaults to None.

Return type
Document

Returns
returns a *Document* instance representing the requested document

async with create_docs(*ids=[]*)

Create documents in bulk.

See *Bulk operations*.

Parameters
ids (List[str]) – list of document ids to be created

Return type
AbstractAsyncContextManager[*BulkCreateOperation*]

Returns
A context manager for the bulk operation

await delete()

Delete the database on the server

Send the request to delete the database and all of its documents.

Return type
None

async for ... in docs(*ids=None*, *create=False*, *prefix=None*, *include_ddocs=False*, ***params*)

A generator to iterator over all or a subset of documents in the database.

When neither of `ids` nor `prefix` are specified, all documents will be iterated. Only one of `ids` and `prefix` can be specified. By default, design documents are not included.

Parameters

- **ids** (Optional[List[str]]) – Allows to iterate over a subset of documents by passing a list of document ids
- **create** (bool) – If `True`, every document contained in *ids*, which doesn't exist, will be represented by an empty *Document* instance.
- **prefix** (Optional[str]) – Allows to iterator over a subset of documents by specifying a prefix that the documents must match.

- **include_ddocs** (bool) – Include the design documents of the database.
- **params** (Any) – Additional query parameters, see CouchDB view endpoint.

Return type
AsyncGenerator[*Document*, None]

async for ... in find(*selector*, *limit=None*, ***params*)

Fetch documents based on search criteria

This method allows to use the _find endpoint of the database.

This method supports all request parameters listed in _find.

Note: As this method returns *Document* s, which contain the complete data, the *fields* parameter is not supported.

Parameters
selector (*type*) – See selectors

Return type
AsyncGenerator[*Document*, None]

Returns
return all documents matching the passed selector.

await get(*id*, *default=None*, ***, *rev=None*)

Returns the document with the given id

Raises

- ***NotFoundError*** – if the given document does not exist and *default* is *None*
- ***BadRequestError*** – if the given rev of the document is invalid or missing

Parameters

- **id** (str) – the name of the document
- **default** (Optional[Dict[str, Any]]) – if *default* is not *None* and the document does not exist on the server, a new *Document* instance, containing *default* as its contents, is returned. To create the document on the server, *save()* has to be called on the returned instance.
- **rev** (Optional[str]) – The requested rev of the document. The requested rev might not or not anymore exist on the connected server.

Return type
Document

Returns
a local representation of the requested document

await index(*index*, ***kwargs*)

Create a new index on the database

This method allows to use the _index endpoint of the database.

This method supports all request parameters listed in _index.

Parameters

- **index** (Dict[str, Any]) – JSON description of the index

- **kwargs** (Any) – additional parameters, refer to the CouchDB documentation

Return type
Dict[str, Any]

Returns
The response of the CouchDB _index endpoint

await info()

Returns basic information about the database

See also GET /{db}.

Returns
Description of returned object.

Return type
def

async with update_docs(*ids=[]*, *create=False*)

Update documents in bulk.

See *Bulk operations*.

Parameters

- **ids** (List[str]) – list of affected documents, defaults to []
- **create** (bool) – [description], defaults to False

Return type
AbstractAsyncContextManager[*BulkUpdateOperation*]

Returns
A context manager for the bulk operation

async for ... in values(***params*)

Iterates over documents in the database

See *docs()*.

Return type
AsyncGenerator[*Document*, None]

class aiocouch.event.**BaseChangeEvent**(*json*)

The base event for shared properties

property id: str

Returns the id of the document

json: Dict[str, Any]

The raw data of the event as JSON

property rev: str

Returns the new rev of the document

property sequence: str

Returns the sequence identifier of the event

class aiocouch.event.**ChangedEvent**(*json*, *database*)

This event denotes that the document got modified

database: *Database*

The database for reference

await doc()

Returns the document after the change

If the `include_docs` was set, this will use the data provided in the received event. Otherwise, the document is fetched from the server.

Return type
Document

class aiocouch.event.**DeletedEvent**(*json*)

This event denotes that the document got deleted

json: Dict[str, Any]

The raw data of the event as JSON

8.4 Documents

A key role in *aiocouch* takes the *Document* class. Every data send and retrieved from the server is represented by an instance of that class. There are no other ways in *aiocouch* to interact with documents.

8.4.1 Getting a Document instance

While the constructor can be used to get an instance representing a specific document, the canonical way is the usage of member functions of instances of the *Database* class.

```
butterfly_doc = await database["butterfly"]
wolpertinger = await database.get("wolpertinger")
```

These methods create a *Document* and fetch the data from the server. For some cases, though, a precise control other the performed requests are required. The above code snippet is equivalent to this:

```
butterfly_doc = Document(database, "butterly")
await butterfly_doc.fetch()
```

8.4.2 Creating new Documents

The creation of a new document on the server consists of three steps. First, you need a local document handle, i.e., an *Document* instance. Then you set the contents of the document. And finally, the local document is saved to the server.

```
# get an Document instance
doc = await database.create("new_doc")

# set the document content
doc["name"] = "The new document"

# actually perform the request to save the document on the server
await doc.save()
```

8.4.3 Modify existing documents

The modification of an existing document works very similarly to the creation. Retrieving the document, updating its contents, and finally saving the modified data to the server.

```
# get an Document instance
doc = await database["existing_doc"]

# update the document content
doc["name"] = "The modified document"

# actually perform the request to save the modification to the server
await doc.save()
```

8.4.4 Using Async Context Managers

To simplify the process of retrieving a document from remote server (or creating a new one if it didn't exist before), modifying it, and saving changes on remote server, you can also use asynchronous context managers.

Using context managers saves you from having to manually perform a lot of these operations as the context managers handle these operations for you automatically.

aiocouch provides async context managers for both *Document* and `SecurityDocument`.

Document Context Manager Example

```
from aiocouch import CouchDB
from aiocouch.document import Document

async with CouchDB(SERVER_URL, USER, PASSWORD) as client:
    # Create database on remote server (fetching it if it already exists)
    my_database = await client.create("my_database", exists_ok=True)

    # If document exists, it's fetched from the remote server
    async with Document(my_database, "secret_agents") as document:
        # Changes are made locally
        document["name"] = "James Bond"
        document["code"] = "007"
    # Upon exit from above context manager, document is saved remotely

    # Display the newly created document after fetching from remote server
    document = await my_database["secret_agents"]
    print(document)
```

Warning: Uncaught exceptions inside the `async with` block will prevent your document changes from being saved to the remote server.

Security Document Context Manager Example

Similarly, you can also use Security Document context manager to add or remove admins or members from a CouchDB database

```
from aiocouch import CouchDB
from aiocouch.document import Document

async with CouchDB(SERVER_URL, USER, PASSWORD) as client:
    # Create database on remote server (fetching it if it already exists)
    my_database = await client.create("my_database", exists_ok=True)

    async with SecurityDocument(my_database) as security_doc:
        # Give user 'bond' member access to 'my_database' database
        security_doc.add_member("bond")
        # Give user 'fleming' admin access to 'my_database' database
        security_doc.add_admin("fleming")
    # Upon exit from above context manager, document is saved remotely

    # Display the recent changes made to security document
    security_doc = await my_database.security()
    print(security_doc)
```

Warning: Uncaught exceptions inside the `async with` block will prevent your security document changes from being saved to the remote server.

8.4.5 Conflict handling

Whenever, two or more different *Document* instances want to save the same document on the server, a *ConflictError* can occur. To cope with conflicts, there are a set of different strategies, which can be used.

One trivial solution is to simply ignore conflicts.This is a viable strategy if only the existance of the document matters.

```
with contextlib.suppress(aiocouch.ConflictError):
    await doc.save()
```

Another straight-forward solution is to override the contents of the existing document. Though, this example code isn't a complete solution either, as the second call to *save()* might raise a *ConflictError* again.

```
try:
    await doc.save()
except aiocouch.ConflictError:
    info = await doc.info()
    doc.rev = info["rev"]
    await doc.save()
```

Other use cases may require a more sophisticated merging of documents. However, there isn't a generic solution to such an approach. Thus, we forego to show example code here.

8.4.6 Reference

class aiocouch.document.**Document**(*database*, *id*, *data=None*)

A local representation for the referenced CouchDB document

An instance of this class represents a local copy of the document data on the server. This class behaves like a dict containing the document data and allows to *fetch()* and *save()* documents. For details about the dict-like interface, please refer to the Python manual.

Constructing an instance of this class does not cause any network requests.

Variables
id – the id of the document

Parameters

- **database** (Database) – The database of the document
- **id** (str) – the id of the document
- **data** (Optional[Dict[str, Any]]) – the initial data used to set the body of the document

attachment(*id*)

Returns the attachment object

The attachment object is returned, but this method doesn't actually fetch any data from the server. Use *fetch()* and *save()*, respectively.

Parameters
id (str) – the id of the attachment

Return type
Attachment

Returns
Returns the attachment object

await copy(*new_id*)

Create a copy of the document on the server

Creates a new document with the data currently stored on the server.

Note: This method uses the COPY /{db}/{docid} endpoint.

If you need to know the *rev* of the created document, use the *Etag* header entry.

Parameters
new_id (str) – the id of the new document

Return type
HTTPResponse

Returns
If the request succeeded, returns the *HTTPResponse* instance.

property data: Optional[Dict[str, Any]]

Returns the document as a dict

If *exists()* is False, this function returns None.

This method does not perform a network request.

Returns
Returns the data of the document or `None`

`await delete`(*discard_changes=False*)

Marks the document as deleted on the server

Calling this method deletes the local data and marks document as deleted on the server. Afterwards, the instance can be filled with new data and call *save()* again.

Note: This method uses the `DELETE /{db}/{docid}` endpoint.

If you want to remove the data from the server, you'd need to use the _purge endpoint instead.

Raises

- ***ConflictError*** – if the local data has changed without saving
- ***ConflictError*** – if the local revision is different from the server. See *Conflict handling*.

Return type
HTTPResponse

Returns
If the request succeeded, returns the *HTTPResponse* instance.

`property exists: bool`

Denotes whether the document exists

A document exists, if an existing was *fetch()* ed from the server and retrieved data doesn't contain the *_deleted* field. Or a new document was saved using *save()*.

This method does not perform a network request.

Returns
`True` if the document exists, `False` overwise

`await fetch`(*discard_changes=False*, *, *rev=None*)

Retrieves the document data from the server

Fetching the document will retrieve the data from the server using a network request and update the local data.

Raises

- ***ConflictError*** – if the local data has changed without saving
- ***BadRequestError*** – if the given rev is invalid or missing

Parameters

- **discard_changes** (`bool`) – If set to *True*, the local data object will the overridden with the retrieved content. If the local data was changed, no exception will be raised.
- **rev** (`Optional[str]`) – The requested rev of the document. The requested rev might not or not anymore exist on the connected server.

Return type
`None`

await info()

Returns a short information about the document.

This method sends a request to the server to retrieve the current status.

Raises
NotFoundError – if the document does not exist on the server

Return type
Dict[str, Any]

Returns
A dict containing the id and revision of the document on the server

property json: Dict[str, Any]

Returns the document content as a JSON-like dict

In particular, all CouchDB-internal document keys will be omitted, e.g., `_id`, `_rev` If *exists()* is `False`, this function returns an empty dict.

This method does not perform a network request.

property rev: Optional[str]

Allows to set and get the local revision

If the local document wasn't fetched or saved, this is `None`.

await save()

Saves the current state to the CouchDB server

Only sends a request, if the local state has been changed since the retrieval of the document data.

Raises
ConflictError – if the local revision is different from the server. See *Conflict handling*.

Return type
Optional[*HTTPResponse*]

Returns
If a successful request was made, returns the *HTTPResponse* instance.

class aiocouch.remote.**HTTPResponse**(*resp*)

Represents an HTTP response from the CouchDB server.

property etag: Optional[str]

Convenient property to access the ETag header in a usable format

headers: Dict[str, str]

The HTTP headers of the response

status: int

The HTTP response status, usually 200, 201 or 202

8.5 Attachments

Attachments are independent binary data attached to a document. They are file-like and require a name and the content type. As attachments do not have size restrictions, they are handled a bit differently than documents in the *Document* class. The content of the attachment isn't cached in the instance at any point, thus data access require a network request.

8.5.1 Getting an Attachment instance

Given a document instance, we can get an Attachment instance using the *attachment()* member function. Unlike with *Document* instances, no data is retrieved from the sever yet.

```
butterfly = await database["butterfly"]
image_of_a_butterfly = butterfly.attachment("image.png")
```

8.5.2 Retrieving the Attachment content

To actually retrieve the data stored on the server, you have to use the *fetch()* method. Once the fetch method is called, the *content_type* member will be set to appropriate value passed from the server.

```
data = await image_of_a_butterfly.fetch()
```

8.5.3 Saving the content of an attachment

8.5.4 Reference

class aiocouch.attachment.**Attachment**(*document*, *id*)

A local representation for the referenced CouchDB document attachment

An instance of this class represents a local copy of an attachment of CouchDB documents.

Variables

- **id** – the id of the attachment
- **content_type** – the content type of the attachment, only available after *fetch()* has been called.

Parameters

- **document** (Document) – The correlated document
- **id** (str) – the id of the attachment

await delete()

Deletes the attachment from the server

Return type
None

await exists()

Checks if the attachment exists on the server

Return type
bool

Returns
returns True if the attachment exists

`await fetch()`

Returns the content of the attachment

Return type
bytes

Returns
the attachment content

`await save(data, content_type)`

Saves the given attachment content on the server

Parameters

- **data** (bytes) – the content of the attachment
- **content_type** (str) – the content type of the given data. (See Content type)

Return type
None

8.6 Design docs and views

The interface for design documents and views aren't final yet.

8.7 Bulk operations

Bulk operations are helpful when you need to create or update several documents within one *Database* with a low amount of requests. In particular, the _bulk_docs endpoint allows to write a bunch of documents in one request.

Bulk operations in *aiocouch* are similar to transactions. You define the set of affected *Document*, apply the changes and finally perform the bulk request. Depending on the particular task, you need to use one of two context manager classes.

For example, the following code affects the documents *foo* and *baz*, existing or not, and sets the key *llama* to *awesome* with one bulk request.

```
async with database.update_docs(["foo", "baz"], create=True) as bulk:
    async for doc in bulk:
        doc["llama"] = "awesome"
```

8.7.1 Include documents in bulk operations

Affected documents can be defined in two ways. The first way is to pass a list of document ids as the *ids* parameter.

```
async with database.update_docs(ids=["foo", "baz"]) as bulk:
    ...
```

The second method is the usage of the *append()* method. Just pass an instance of *Document* and its content will be saved as part of the bulk operation.

```
the_document = Document(...)

async with BulkOperation(database=my_database) as bulk:
    bulk.append(the_document)
```

Once the control flow leaves the context, the bulk operation persists the applied changes to all documents that there included in the bulk operation one or the other way. Also, both ways can be mixed.

8.7.2 Create many documents in one operation

To create many documents, you use the *create_docs()* method to get the context manager. Include documents as described above. Once the context manager closes, one request containing all document contents gets send to the server.

```
async with my_database.create_docs(...) as bulk:
    for doc in bulk:
        # make changes to the Document instances

# the request was send now
```

Note that the bulk operation does not check, if the requested documents alrady exists on the server. Instead, the *error* list will contain *conflict* in the *error* field corresponding to the document.

8.7.3 Update many documents in one operation

To update many documents, you use the *update_docs()* method to get the context manager. Include documents as described above. Once the context manager closes, one request containing all document contents gets send to the server. In contrast to the create operation, the *BulkUpdateOperation* context manager will request all documents whose ids where passed as the *ids* parameter. If you already have *Document* instance, you may want to use the *append()* method instead.

```
my_doc: Document = ...

async with my_database.update_docs(...) as bulk:
    bulk.append(my_doc)

    for doc in docs:
        # make changes to the Document instances

# the request was send now
```

8.7.4 Error handling for bulk operations

The important bit first, none of the bulk operation context manager will raise an exception if something in the request went wrong. Each individual document can be saved successfully or may have an error. It's in your responsibility to check the status after the request finished.

You can check the status of each document with the *ok*, *error*, and *response* properties of the context manager. The *ok* and *error* lists contain all documents that could and couldn't be saved properly, respectively. The *response* contains the response from the CouchDB server. So in case of an error, it will contain a description of what went wrong.

```
async with BulkOperation(database=my_database) as bulk:
    ...

if len(bulk.error) == 0:
    print(f"Saved all {len(bulk.ok)} documents")
else:
    print(f"Failed to saved {len(bulk.error)} documents")
```

8.7.5 Reference

class aiocouch.bulk.**BulkOperation**(*database*)

A context manager for bulk operation. This operation allows to write many documents in one request.

Bulk operations use the _bulk_docs endpoint of the database.

To populate the list of written documents, use the *append()* method.

Parameters
database (*Database*) – The database used in the bulk operation

async for ... in __aiter__()

An iterator that yields Document instances that are part of this bulk operation.

Returns
Every *Document* instance that will be affected by this operation

Return type
AsyncGenerator[*Document*, None]

append(*doc*)

Add a document to this bulk operation.

Parameters
doc (*Document*) – the document that should be stored as part of the bulk operation

Raises
ValueError – if the provided document instance is already part of the bulk operation

Return type
Document

Returns
the provided document

create(*id*, *data=None*)

Create a new document as part of the bulk operation

Parameters

- **id** (str) – the id of the document
- **data** (Optional[Dict[str, Any]]) – the inital data used to set the body of the document, defaults to None

Raises
ValueError – if the provided document id is already part of the bulk operation

Return type
Document

Returns
a Document instance reference the newly created document

error: Optional[List[*Document*]]

The list of all *Document* instances that could not be saved to server.

Only available after the context manager has finished without a passing exception.

ok: Optional[List[*Document*]]

The list of all *Document* instances that there successfully saved to server.

Only available after the context manager has finished without a passing exception.

response: Optional[List[Dict[str, Any]]]

The resulting JSON response of the _bulk_docs request. Refer to the CouchDB documentation for the contents.

Only available after the context manager has finished without a passing exception.

property status: Optional[List[Dict[str, Any]]]

Deprecated since version 2.1.0: Use the response property instead.

update(*doc*)

Add a document to this bulk operation.

Parameters
doc (*Document*) – the document that should be stored as part of the bulk operation

Raises
ValueError – if the provided document instance is already part of the bulk operation

Return type
Document

Returns
the provided document

Deprecated since version 2.1.0: Use append(doc) instead. It just makes more sense.

class aiocouch.bulk.**BulkCreateOperation**(*database*, *ids=[]*)

A context manager for bulk creation operations. This operation allows to write many documents in one request.

Bulk operations use the _bulk_docs endpoint of the database.

Parameters

- **database** (*Database*) – The database used in the bulk operation
- **ids** (List[str]) – a list of ids of the involved documents, defaults to []

error: Optional[List[*Document*]]

The list of all *Document* instances that could not be saved to server.

Only available after the context manager has finished without a passing exception.

ok: Optional[List[*Document*]]

The list of all *Document* instances that there successfully saved to server.

Only available after the context manager has finished without a passing exception.

response: Optional[List[Dict[str, Any]]]

The resulting JSON response of the _bulk_docs request. Refer to the CouchDB documentation for the contents.

Only available after the context manager has finished without a passing exception.

class aiocouch.bulk.**BulkUpdateOperation**(*database*, *ids=[]*, *create=False*)

A context manager for bulk update of documents. In particular, for every provided id, a *Document* instance is provided. The data is fetched using the `AllDocsView` with a minimal amount of requests.

Parameters

- **database** (*Database*) – The database of the bulk operation
- **ids** (List[str]) – list of document ids
- **create** (bool) – If `True`, every document contained in *ids* that doesn't exist, will be represented by an empty *Document* instance.

error: Optional[List[*Document*]]

The list of all *Document* instances that could not be saved to server.

Only available after the context manager has finished without a passing exception.

ok: Optional[List[*Document*]]

The list of all *Document* instances that there successfully saved to server.

Only available after the context manager has finished without a passing exception.

response: Optional[List[Dict[str, Any]]]

The resulting JSON response of the _bulk_docs request. Refer to the CouchDB documentation for the contents.

Only available after the context manager has finished without a passing exception.

8.8 Exceptions

Most errors you encounter in aiocouch stem from HTTP request to the CouchDB server. Most of those are therefore captured an transformed into exceptions. There might still be other errors, however those should not be encountered under normal operation.

For further details, what can cause individual status codes, see also HTTP Status codes.

exception aiocouch.**BadRequestError**

Represents a 400 HTTP status code returned from the server

exception aiocouch.**ConflictError**

Represents a 409 HTTP status code returned from the server

exception aiocouch.**ExpectationFailedError**

Represents a 417 HTTP status code returned from the server

exception aiocouch.**ForbiddenError**

Represents a 403 HTTP status code returned from the server

exception aiocouch.**NotFoundError**

Represents a 404 HTTP status code returned from the server

exception aiocouch.**PreconditionFailedError**

Represents a 412 HTTP status code returned from the server

exception aiocouch.**UnauthorizedError**

Represents a 401 HTTP status code returned from the server

exception aiocouch.**UnsupportedMediaTypeError**

Represents a 415 HTTP status code returned from the server

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J