Rapid Dev Storage

Release 1.0.0

Michael Hall

May 09, 2020

REFERENCE:

CHAPTER

ONE

CLASS REFERENCE

class rapid_dev_storage._base_api.**Storage**(*backend:* rapid_dev_storage._types.StorageBackend)

This is the basic storage wrapper class.

It can be extended with additional functionality and adapters for specific data types, as well as adding facotry methods for instantiation including the required backend

class rapid_dev_storage._base_api.**StorageGroup**(*backend:* rapid_dev_storage._types.StorageBackend, *group_name: str*)

async for ... in all_items() → AsyncIterator[Tuple[Tuple[str, ...], Union[Dict[str, Any], List[Any], int, float, str, None]]]

Iterates over all items stored in the group

The data is yielded as a 2-tuple consisting of the tuple key, and the value which was associated

clear_group() → Awaitable[None]

Clears an entire group

class rapid_dev_storage._base_api.**StoredValue**(*backend:* rapid_dev_storage._types.StorageBackend, *group_name: str*, **keys: str*)

Representation of everything needed to interact with a stored value, methods included

clear_value() → Awaitable[None]

Clears a value. This does not require that the value already existed

get_value() → Awaitable[Union[Dict[str, Any], List[Any], int, float, str, None, rapid_dev_storage._types._NoValueType]]

Gets a value if it exists, otherwise returns `NoValue`

set_value(*value: Union[Dict[str, Any], List[Any], int, float, str, None]*) → Awaitable[None]

Sets a value

class rapid_dev_storage._sqlite_backend.**SQLiteBackend**(*connection*, *table_name: str*, *serializer*, *deserializer*)

This holds all the SQLite Logic.

All lookups operate on a composite primary key, ensuring that the abstration has minimal runtime performance overhead. This does incur a small cost to the DB size, though this is acceptible.

Interface is async despite the underlying code not being so. This is intentional, as if used as-is, without competeting on the same table with other applications, it should not block the event loop.

Meanwhile, the interface being async consistently leaves room for drop in replacements which may actually utilize the async nature, or further features which might have the potential to be blocking

There are a handful of computed SQL queries. These are limited against user input, and userinput is not allowed to be formatted in, with 1 exception of the table name. This name is restricted in nature as to be safe, and properly bracketed so that it is never seen as an SQL expression

Changes to the computed queries should be done with caution to ensure this remains true. For additional peace of mind, you can choose to disallow user input from being used as part of the table_name at the application layer, which leaves all remaining potential user input inserted as parameters.

async for ... in get_all_by_group(*group_name: str*)
Concrete implmentations must asynchronously yield a 2-tuple of (key tuple, value)

async for ... in get_all_by_key_prefix(*group_name: str*, **keys: str*)
Concrete implementations must asynchronously yield a 2-tuple of (key tuple, value)

class rapid_dev_storage._types.**StorageBackend**
This abstract base class shows the interfaces required to use a class as a replacement backend for the included ones

Interfaces here are async to allow dropping in other interfaces which would strictly need to be async

abstractmethod get_all_by_group(*group_name: str*) → AsyncIterator[Tuple[Tuple[str, ...], Union[Dict[str, Any], List[Any], int, float, str, None]]]
Concrete implmentations must asynchronously yield a 2-tuple of (key tuple, value)

abstractmethod get_all_by_key_prefix(*group_name: str*, **keys: str*) → AsyncIterator[Tuple[Tuple[str, ...], Union[Dict[str, Any], List[Any], int, float, str, None]]]
Concrete implementations must asynchronously yield a 2-tuple of (key tuple, value)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX