awstin

Release 0.0.22

Kevin Duff

Apr 14, 2021

CONTENTS

CHAPTER

ONE

USER DOCUMENTATION

1.1 API Gateway

1.1.1 Auth Lambdas

Authorizor lambda responses can be generated with helper functions provided by *awstin.apigateway.auth*. *awstin.apigateway.auth.accept()*, *awstin.apigateway.auth.reject()*, *awstin.apigateway.auth.unauthorized()*, and *awstin.apigateway.auth.invalid()* will produce properly formatted auth lambda responses.

```
from awstin.apigateway import auth


def auth_event_parser(event, _context):
    token = event["headers"]["AuthToken"]
    resource_arn = event["methodArn"]
    principal_id = event["requestContext"]["connectionId"]

    return token, resource_arn, principal_id


@lambda_handler(auth_event_parser)
def token_auth(token, resource_arn, principal_id):
    if token == "good token":
        return auth.accept(principal_id, resource_arn)
    elif token == "bad token":
        return auth.reject(principal_id, resource_arn)
    elif token == "unauthorized token":
        return auth.unauthorized()
    else:
        return auth.invalid()
```

1.1.2 Serverless Websockets

Websocket pushes can be performed with a callback URL and message:

```
from awstin.apigateway.websocket import Websocket


Websocket("endpoint_url", stage="dev").send("callback_url", "message")
```

1.2 Lambdas

1.2.1 Writing Lambda Handlers

Lambda handlers can be wrapped with the *awstin.awslambda.lambda_handler()* decorator factory, which accepts a parser function as an argument. The parser should accept an AWS event and context, and should return inputs to the wrapped function as a tuple (to be passed in as args) or dict (to be passed in as kwargs).

```
from awstin.awslambda import lambda_handler

def event_parser(event, context):
    request_id = event["requestContext"]["requestId"]
    memory_limit = context["memory_limit_in_mb"]
    return request_id, memory_limit


@lambda_handler(event_parser)
def handle_custom_event(request_id, memory_limit):
    print(request_id)
    print(memory_limit)
```

In this way, the event parsing and business logic of Lambda handlers are kept separate.

1.2.2 Testing Lambda Handlers

The way `awstin` separates Lambda handlers into a parser and a main function promotes testing the function in parts as well.

The parser can be tested individually given example events, and asserting that the returned values are expected inputs to the main function.

```
def my_parser(event, context):
    return event["a"], event["b"]

@lambda_handler(my_parser)
def my_handler(a: int, b: str):
    ...

# -------

def test_parser():
    args = my_parser(test_event, test_context)
    assert ...
```

The main function can be tested directly as well. When a function `my_handler` is wrapped with *awstin.awslambda.lambda_handler()*, the returned function has an `inner` attribute containing the wrapped function.

```
def test_handler():
    result = my_handler.inner(1, "abc")
    assert ...
```

1.3 DynamoDB

1.3.1 Getting started with DynamoDB in `awstin`

This tutorial follows the "Getting Started" guide for DynamoDB and Python in the `boto3` docs. That reference can be found here.

Equivalent tutorials are presented for each section, except for the first and last sections on creating and deleting tables. These functions of `boto3` are out-of-scope for `awstin`, leaving infrastructure management to dedicated IaC frameworks or to `boto3`.

In each case, the structure of the examples are kept as comparable as possible to the examples in the `boto3` docs.

Note that when using DynamoDB in `awstin`, either the `AWS_REGION` (in production) or the `TEST_DYNAMODB_ENDPOINT` (in integration tests) environment variable should be set. This will be used to connect either to the real AWS DynamoDB service or to a testing instance of DynamoDB.

Defining Data Models

The elemental object for DynamoDB in `awstin` is not the JSON dict, but instead a *awstin.dynamodb.DynamoModel* subclass representing a view of the data in a table or index.

The `awstin.dynamodb.DynamoModel` Class

Table Models

The elemental object for DynamoDB in `awstin` is not the JSON dict, but instead a *awstin.dynamodb.DynamoModel* subclass representing a view of the data in a table or index.

Each *awstin.dynamodb.DynamoModel* subclass should have a `_table_name_` attribute on its class definition which is the name of the table in DynamoDB the model relates to.

The class should also have definitions of table keys and any additional attributes you want through *awstin.dynamodb.Key* and *awstin.dynamodb.Attr* definitions. By default, the attribute name on the data model is the property name in DynamoDB, but a property name can also be specified by passing in a string argument.

Below is a data model representing information for the Movies table in the AWS documentation example.

```
from awstin.dynamodb import Attr, DynamoModel, Key


class Movie(DynamoModel):
    _table_name_ = "Movies"

    #: Year the film  was made (hash key)
```

(continues on next page)

(continued from previous page)

```
    year = Key()

    #: Title of the film (sort key)
    title = Key()

    #: Additional information about the film
    info = Attr()
```

Index Models

awstin.dynamodb.DynamoModel subclasses can also reference local or global secondary indexes. These work the same as table data models, but in addition to the `_table_name_` attribute, an `_index_name_` attribute should also be provided, defining the name of the index.

Accessing DynamoDB Tables

Once data models are defined, they can be used to interact with DynamoDB tables. This is via the *awstin.dynamodb.DynamoDB* class, which connects to DynamoDB either via the `AWS_REGION` (in production) or the `TEST_DYNAMODB_ENDPOINT` (in integration tests) environment variable. Tables are accessed from the *awstin.dynamodb.DynamoDB* instance via indexing by *awstin.dynamodb.DynamoModel* subclasses.

```
from awstin.dynamodb import DynamoDB


dynamodb = DynamoDB()
table = dynamodb[Movie]
```

Loading Sample Data

awstin.dynamodb.Table.put_item() takes instances of the data model to insert into DynamoDB after serialization.

`awstin` handles conversions between `float` and `Decimal` internally, so `float` can be used when instantiating data models.

```
import json

from models import Movie

from awstin.dynamodb import DynamoDB


def load_movies(movies):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    for movie_json in movies:
        movie = Movie(
            title=movie_json["title"],
            year=movie_json["year"],
            info=movie_json["info"],
```

(continues on next page)

(continued from previous page)

```
        )
        table.put_item(movie)


if __name__ == "__main__":
    with open("moviedata.json", "r") as json_file:
        movie_list = json.load(json_file)
    load_movies(movie_list)
```

awstin.dynamodb.DynamoModel has *awstin.dynamodb.DynamoModel.serialize()* and *awstin.dynamodb.DynamoModel.deserialize()* methods to convert between DynamoDB representations of data and the data models, and can be used to read data from JSON.

```
import json

from models import Movie

from awstin.dynamodb import DynamoDB


def load_movies(movies):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    for movie in movies:
        table.put_item(movie)


if __name__ == "__main__":
    with open("moviedata.json", "r") as json_file:
        movie_list = json.load(json_file)
    load_movies([Movie.deserialize(movie) for movie in movie_list])
```

CRUD Operations

This section goes over CRUD operations on the table via the data models outlined in the previous sections.

Create a New Item

As demonstrated in the last section, new items are added by instantiating the data model classes and passing them to *awstin.dynamodb.Table.put_item()*.

```
from models import Movie

from awstin.dynamodb import DynamoDB


def put_movie(title, year, plot, rating):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    movie = Movie(
        title=title,
```

(continues on next page)

(continued from previous page)

```
        year=year,
        info={
            "plot": plot,
            "rating": rating,
        },
    )
    response = table.put_item(movie)
    return response


if __name__ == "__main__":
    movie_resp = put_movie(
        "The Big New Movie",
        2015,
        "Nothing happens at all.",
        0,
    )
    print("Put movie succeeded:")
    print(movie_resp)
```

Read an Item

Items can be retrieved from a table in a dict-like style. If the table only has a hash key, items are accessed from the value of the hash key. If it has a hash key and a sort key, they're accessed by a tuple of (hash key value, sort key value).

```
from models import Movie

from awstin.dynamodb import DynamoDB


def get_movie(title, year):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    item: Movie = table[year, title]

    return item


if __name__ == "__main__":
    movie = get_movie("The Big New Movie", 2015)
    if movie:
        print("Get movie succeeded:")
        print(type(movie))
        print(movie.serialize())
```

Update an Item

`awstin` provides an update expression syntax that allows update expressions to be built and chained together with `&`. The *awstin.dynamodb.Attr.set()*, *awstin.dynamodb.Attr.remove()*, *awstin.dynamodb.Attr.add()*, and *awstin.dynamodb.Attr.delete()* methods correspond to the update operations available in DynamoDB.

awstin.dynamodb.Table.update_item() takes the primary key as the first argument.

```
from models import Movie

from awstin.dynamodb import DynamoDB


def update_movie(title, year, rating, plot, actors):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    update_expression = (
        Movie.info.rating.set(rating)
        & Movie.info.plot.set(plot)
        & Movie.info.actors.set(actors)
    )

    return table.update_item(
        key=(year, title),
        update_expression=update_expression,
    )


if __name__ == "__main__":
    update_response = update_movie(
        "The Big New Movie",
        2015,
        5.5,
        "Everything happens all at once.",
        ["Larry", "Moe", "Curly"],
    )
    print("Update movie succeeded:")
    print(update_response.serialize())
```

Increment an Atomic Counter

awstin.dynamodb.Attr.set() can take expressions defining the new value.

```
from models import Movie

from awstin.dynamodb import DynamoDB


def increase_rating(title, year, rating_increase):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    response = table.update_item(
        (year, title),
```

(continues on next page)

(continued from previous page)

```
        update_expression=Movie.info.rating.set(Movie.info.rating + rating_increase),
    )
    return response


if __name__ == "__main__":
    update_response = increase_rating("The Big New Movie", 2015, 1)
    print("Update movie succeeded:")
    print(update_response.serialize())
```

Update an Item (Conditionally)

awstin.dynamodb.Attr.set() can optionally be given a condition expression. The updated value will be returned as an instance of the data model. If the update condition fails, `None` will be returned instead.

More information about the query/condition syntax is given in *Querying and Scanning the Data*.

```
from models import Movie

from awstin.dynamodb import DynamoDB


def remove_actors(title, year, actor_count):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    response = table.update_item(
        (year, title),
        update_expression=Movie.info.actors[0].remove(),
        condition_expression=Movie.info.actors.size() >= actor_count,
    )
    return response


if __name__ == "__main__":
    update_response = remove_actors("The Big New Movie", 2015, 3)
    if update_response:
        print("Updated")
        print(update_response.serialize())
    else:
        print("Not Updated")
```

Delete an Item

Items can also be deleted by primary key value. A condition expression can also be provided. If the item is deleted, *awstin.dynamodb.Table.delete_item()* returns `True`. If the condition fails, it returns `False`.

More information about the query/condition syntax is given in *Querying and Scanning the Data*.

```
from models import Movie

from awstin.dynamodb import DynamoDB
```

(continues on next page)

(continued from previous page)

```
def delete_underrated_movie(title, year, rating):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    return table.delete_item(
        (year, title),
        condition_expression=Movie.info.rating <= rating,
    )


if __name__ == "__main__":
    print("Attempting a conditional delete...")
    deleted = delete_underrated_movie("The Big New Movie", 2015, 5)
    if deleted:
        print("Deleted film")
    else:
        print("Did not delete film")
```

Querying and Scanning the Data

A query/condition syntax is provided by `awstin`. This is similar to the syntax provided by `sqlalchemy`, for example. Queries can be built and combined with `&` and `|`.

There are methods on *`awstin.dynamodb.Attr`* and *`awstin.dynamodb.Key`* corresponding to the condition operations provided by `boto3`/DynamoDB, with arithmetic comparisons exposed Pythonically.

Query the Data

`awstin.dynamodb.Table.query()` takes a condition expression for the query, and optionally a post-query scan expression (which is much more permissive).

```
from models import Movie

from awstin.dynamodb import DynamoDB


def query_movies(year):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    return table.query(Movie.year == year)


if __name__ == "__main__":
    query_year = 1985
    print(f"Movies from {query_year}")
    movies = query_movies(query_year)
    for movie in movies:
        print(movie.year, ": ", movie.title)
```

More Detailed Query

Condition expressions can be combined with logical operators `&` and `|`.

```
from models import Movie

from awstin.dynamodb import DynamoDB


def query_and_project_movies(year, title_range):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    query = (Movie.year == year) & (Movie.title.between(*title_range))

    return table.query(query)


if __name__ == "__main__":
    query_year = 1992
    query_range = ("A", "L")
    print(
        f"Get movies from {query_year} with titles from "
        f"{query_range[0]} to {query_range[1]}"
    )
    movies = query_and_project_movies(query_year, query_range)
    for movie in movies:
        print(f"{movie.year} : {movie.title}")
        print(movie.info)
        print()
```

Scan the Data

Scans take much more permissive condition expressions, but are performed after-the-fact on data retrieved from DynamoDB.

```
from models import Movie

from awstin.dynamodb import DynamoDB


def scan_movies(year_range, display_movies):
    dynamodb = DynamoDB()
    table = dynamodb[Movie]

    display_movies(table.scan(Movie.year.between(*year_range)))


if __name__ == "__main__":

    def print_movies(movies):
        for movie in movies:
            print(f"{movie.year} : {movie.title}")
            print(movie.info)
            print()
```

(continues on next page)

(continued from previous page)

```
    query_range = (1950, 1959)
    print(f"Scanning for movies released from {query_range[0]} to {query_range[1]}...
↪")
    scan_movies(query_range, print_movies)
```

1.3.2 Integration Testing `awstin.dynamodb` Projects

Note that when integration testing DynamoDB code in `awstin`, the `TEST_DYNAMODB_ENDPOINT` environment variable should be set to the endpoint of a dockerized DynamoDB instance.

The central tool for building integration tests for projects using DynamoDB in `awstin` is *`awstin.dynamodb.testing.temporary_dynamodb_table()`*. This context manager creates a DynamoDB table with the provided information on entry, and destroys it on exit. It ensures that these operations are completed before entry and exit to prevent any race conditions.

For projects with IaC, integration testing can introduce multiple sources of truth for the infrastructure that can get out-of-sync. To help combat this, `awstin` provides *`awstin.dynamodb.testing.create_serverless_tables()`*. This will create tables based on the given *serverless.yml* file. Note that this functionality is still basic, and can't intepret variables in these resources yet.

It's easy to build fixtures or mixins on top of these context managers to produce tables in whatever state you'd like to emulate production scenarios for testing.

1.4 SNS

SNS topics can be retrieved by name and published to with the message directly. This requires either the `TEST_SNS_ENDPOINT` (for integration testing) or `AWS_REGION` (for production) environment variable to be set.

```
from awstin.sns import SNSTopic


topic = SNSTopic("topic-name")
message_id = topic.publish("a message")
```

Message attributes can also be set from the kwargs of the publish.

```
topic.publish(
   "another message",
   attrib_a="a string",
   attrib_b=1234,
   attrib_c=["a", "b", False, None],
   attrib_d=b"bytes value",
)
```

CHAPTER

TWO

API DOCUMENTATION

2.1 awstin.apigateway package

2.1.1 awstin.apigateway.auth module

awstin.apigateway.auth.**accept**(*principal_id*, *resource_arn*)
Return an auth lambda response granting access to the given resource ARN to the given principle ID.

Parameters

- **principal_id** (*str*) – The principal ID to grant access to
- **resource_arn** (*str*) – The ARN of the resource to grant access to

Returns Auth lambda response

Return type dict

awstin.apigateway.auth.**invalid**(*body='Invalid'*)
Return an auth lambda response indicating the request is invalid.

Parameters **body** (*str, optional*) – Optional resposnse body. Default "Invalid"

Returns Auth lambda response

Return type dict

awstin.apigateway.auth.**reject**(*principal_id*, *resource_arn*)
Return an auth lambda response rejecting access to the given resource ARN to the given principle ID.

Parameters

- **principal_id** (*str*) – The principal ID to reject access to
- **resource_arn** (*str*) – The ARN of the resource to reject access to

Returns Auth lambda response

Return type dict

awstin.apigateway.auth.**unauthorized**(*body='Unauthorized'*)
Return an auth lambda response indicating the requester is unauthorized.

Parameters **body** (*str, optional*) – Optional resposnse body. Default "Unauthorized"

Returns Auth lambda response

Return type dict

2.1.2 awstin.apigateway.websocket module

class awstin.apigateway.websocket.**Websocket**(*domain_name*, *stage=None*)
Bases: object

Serverless-to-client push via websocket

send(*connection_id*, *message*)
Send a message to the user

2.2 awstin.awslambda package

2.2.1 awstin.awslambda module

awstin.awslambda.**lambda_handler**(*event_parser*)
Decorator factory for wrapping a lambda handler in a boilerplate event logger and parser.

The wrapped function is on the returned function's *inner* attribute in order to be available for testing.

Parameters **event_parser** (*callable*) – Parser of a lambda handler input. Should take as an input the event and context as (dict, dict), and return a list of arguments that the wrapped handler should use.

Returns **handler** – Decorator for the Lambda handler, accepting a LambdaEvent. It logs the raw incoming event and the result.

Return type callable

2.3 awstin.dynamodb package

2.3.1 awstin.dynamodb module

class awstin.dynamodb.**Attr**(*attribute_name: Optional[str] = None*)
Bases: awstin.dynamodb.orm.BaseAttribute

Used to define and query non-key attributes on a dynamodb table data model

add(*expression*)
Add to an attribute (numerical add or addition to a set). Corresponds to ADD as part of the update expression in Table.update_item.

Parameters **expression** (*UpdateOperand*) – Value to add

attribute_type(*value*)
Filter results by attribute type

Parameters **value** (*str*) – Index for a DynamoDB attribute type (e.g. "N" for Number)

begins_with(*value*)
Filter results by a key or attribute beginning with a value

Parameters **value** (*str*) – Starting string for returned results

between(*low*, *high*)
Filter results by range (inclusive)

Parameters

- **low** (*Any*) – Low end of the range
- **high** (*Any*) – High end of the range

contains(*value*)
Filter results by attributes that are containers and contain the target value

Parameters **values** (*Any*) – Result must contain this item

delete(*expression*)
Delete part of a set attribute. Corresponds to DELETE as part of the update expression in `Table.update_item`.

Parameters **expression** (*UpdateOperand*) – Value to delete

exists()
Filter results by existence of an attribute

if_not_exists(*value*)
Conditionally return a value if this attribute doesn't exist on the model

in_(*values*)
Filter results by existence in a set

Parameters **values** (*list of Any*) – Allowed values of returned results

not_exists()
Filter results by non-existence of an attribute

remove()
Remove an attribute. Corresponds to REMOVE as part of the update expression in `Table.update_item`.

set(*expression*)
Set an attribute to a new value. Corresponds to SET as part of the update expression in `Table.update_item`.

Parameters **expression** (*UpdateOperand*) – New value, or an expression defining a new value

size()
Filter by size of a collection

class awstin.dynamodb.**DynamoDB**(*timeout=5.0*, *max_retries=3*)
Bases: `object`

A client for use of DynamoDB via awstin.

Tables are accessed via data models. See documentation for details.

list_tables()
Return a list of all table names in this DynamoDB instance.

Returns Table names

Return type list of str

class awstin.dynamodb.**DynamoModel**(***kwargs*)
Bases: `object`

Class defining an ORM model for a DynamoDB table.

Subclasses must have a `_table_name_` attribute. Attributes making up the data model should be Attr or Key instances.

Subclasses representing indexes should also have an `_index_name_` attribute

classmethod deserialize(*data*)
Deserialize JSON into a DynamoModel subclass. Internally converts Decimal to float in the deserialization.

Parameters **data** (*dict of (str, Any)*) – Serialized model

Returns The deserialized data model

Return type *DynamoModel*

serialize()
Serialize a DynamoModel subclass to JSON that can be inserted into DynamoDB. Internally converts float to Decimal.

Returns The serialized JSON entry

Return type dict of (str, Any)

class awstin.dynamodb.**Key**(*attribute_name: Optional[str] = None*)
Bases: awstin.dynamodb.orm.BaseAttribute

Used to define and query hash and sort key attributes on a dynamodb table data model

add(*expression*)
Add to an attribute (numerical add or addition to a set). Corresponds to ADD as part of the update expression in Table.update_item.

Parameters **expression** (*UpdateOperand*) – Value to add

attribute_type(*value*)
Filter results by attribute type

Parameters **value** (*str*) – Index for a DynamoDB attribute type (e.g. "N" for Number)

begins_with(*value*)
Filter results by a key or attribute beginning with a value

Parameters **value** (*str*) – Starting string for returned results

between(*low*, *high*)
Filter results by range (inclusive)

Parameters

- **low** (*Any*) – Low end of the range
- **high** (*Any*) – High end of the range

contains(*value*)
Filter results by attributes that are containers and contain the target value

Parameters **values** (*Any*) – Result must contain this item

delete(*expression*)
Delete part of a set attribute. Corresponds to DELETE as part of the update expression in Table.update_item.

Parameters **expression** (*UpdateOperand*) – Value to delete

exists()
Filter results by existence of an attribute

if_not_exists(*value*)
Conditionally return a value if this attribute doesn't exist on the model

in_(*values*)
Filter results by existence in a set

Parameters **values** (*list of Any*) – Allowed values of returned results

not_exists()
Filter results by non-existence of an attribute

remove()
Remove an attribute. Corresponds to REMOVE as part of the update expression in `Table.update_item`.

set(*expression*)
Set an attribute to a new value. Corresponds to SET as part of the update expression in `Table.update_item`.

Parameters **expression** (*UpdateOperand*) – New value, or an expression defining a new value

size()
Filter by size of a collection

class awstin.dynamodb.**Table**(*dynamodb_client*, *data_model*)
Bases: `object`

Interface to a DynamoDB table.

Items can be retrieved from the table by a shorthand depending on the primary key. If it's only a partition key, items can be retrieved by the value of the partition key:

```
my_table["hashval"]
```

If it's a partition and sort key, items can be retrived by a hashkey, sortkey tuple:

```
my_table["hashval", 123]
```

Items can also be retrieved in a dict-like way:

```
my_table[{"HashKeyName":  "hashval", "SortKeyName":  123}]
```

delete_item(*key*, *condition_expression=None*)
Delete an item, given either a primary key as a dict, or given simply the value of the partition key if there is no sort key

Parameters

- **key** (*Any*) – Primary key of the entry to delete, specified as a hash key value, composite key tuple, or a dict
- **condition_expression** (*Query, optional*) – Optional condition expression for the delete, intended to make the operation idempotent

Returns **deleted** – True if the delete, False if the condition was not satisfied

Return type bool

Raises **botocore.exceptions.ClientError** – If there's an error in the request.

put_item(*item*)
Put an item in the table

Parameters **item** (DynamoModel) – The item to put in the table

query(*query_expression*, *filter_expression=None*)
Yield items from the table matching some query expression and optional filter expression. Lazily paginates items internally.

Parameters

- **query_expression** (*Query*) – A Key query constructed with awstin's query syntax
- **filter_expression** (*Query*) – An additional post-query filter expression constructed with awstin's query syntax

Yields **item** (*DynamoModel*) – An item in the table matching thw query

scan(*scan_filter=None*)

Yield items in from the table, optionally matching the given filter expression. Lazily paginates items internally.

Parameters **scan_filter** (*Query*) – An optional query constructed with awstin's query framework

Yields **item** (*DynamoModel*) – An item in the table matching the filter

update_item(*key*, *update_expression*, *condition_expression=None*)

Update an item in the table given an awstin update expression.

Can optionally have a condition expression.

Parameters

- **key** (*Any*) – Primary key, specified as a hash key value, composite key tuple, or a dict
- **update_expression** (*awstin.dynamodb.orm.UpdateOperator*) – Update expression. See docs for construction.
- **condition_expression** (*Query, optional*) – Optional condition expression

Returns Updated model, or None if the condition expression fails

Return type *DynamoModel* or None

awstin.dynamodb.**list_append**(*left*, *right*)

Set a value to the combination of two lists in an update expression

2.3.2 awstin.dynamodb.testing module

awstin.dynamodb.testing.**create_serverless_tables**(*sls_filename: str*, *delay: float = 5.0*, *max_attempts: int = 10*)

Parse a serverless.yml file, deploying any tables found in the resources section.

This is currently very basic functionality that needs fleshing out. See k2bd/awstin#99

Parameters

- **sls_filename** (*str*) – Location of the serverless.yml file
- **delay** (*float, optional*) – Delay in seconds between checks if the table exists
- **max_attempts** (*int, optional*) – Max number of attempts to check if the table exists, after which the client gives up

awstin.dynamodb.testing.**temporary_dynamodb_table**(*data_model*, *hashkey_name*, *hashkey_type='S'*, *sortkey_name=None*, *sortkey_type='S'*, *delay=5.0*, *max_attempts=10*, *extra_attributes=None*, ***extra_kwargs*)

Context manager creating a temporary DynamoDB table for testing.

Ensures that the table is created and destroyed before entering and exiting the context.

Parameters

- **data_model** (DynamoModel) – Model to interface with this table
- **hashkey_name** (*str*) – Name of the hash key of the table
- **hashkey_type** (*str, optional*) – Type of the hash key ("S", "N", or "B"). Default "S"
- **sortkey_name** (*str, optional*) – Optional sort key for the temporary table
- **sortkey_type** (*str, optional*) – Type of the sort key if there is one ("S", "N", or "B"). Default "S"
- **delay** (*float, optional*) – Delay in seconds between checks if the table exists
- **max_attempts** (*int, optional*) – Max number of attempts to check if the table exists, after which the client gives up.
- **extra_attributes** (*dict, optional*) – Additional attribute definitions (boto3 specification)
- ****extra_kwargs** (*dict*) – Additional keyword arguments to pass to create_table

2.4 awstin.sns package

2.4.1 awstin.sns module

class awstin.sns.**SNSTopic**(*topic_name*)

Bases: object

A client for typical use of an SNS topic

publish(*message*, ***attributes*)

Publish a message to the topic

Parameters

- **message** (*str*) – The message to send
- ****attributes** (*dict(str, Any)*) – Message attributes to add to the message. Value must be castable into "String" (str), "String.Array" (list of str), "Number" (int or float), and "Binary" (bytes). If it's not classified, a bytes cast will be attempted.

Returns The message's unique ID

Return type str

2.4.2 awstin.sns.testing module

PYTHON MODULE INDEX

a

INDEX