askhome Documentation

Release 0.1

Matěj Hlaváček

Jun 02, 2017

Contents

Askhome wraps the Smart Home Skill API for Amazon Echo and removes all that ugly boilerplate.

Basic skill in askhome looks like this:

```python
from askhome import Smarthome, Appliance

class Star(Appliance):
    @Appliance.action
    def turn_on(self, request):
        ... # Let there be light

home = Smarthome()
home.add_appliance('star1', Star, name='Sun')

handler = home.lambda_handler
```

CHAPTER 1

Features

- Define what your smart devices can do with simple class interface
- Askhome then handles device discovery and routes Alexa requests to your methods
- Requests come in a nice preprocessed object and provide easy to use response methods
- If something goes wrong, just raise the appropriate exception
- You don't have to touch the raw JSON requests at all!

CHAPTER 2

Why a Smart Home Skill

Alexa Custom Skills are indeed much more flexible, but creating their intent schema can be a hassle. If you want to simply control your devices, Smart Home Skills provide a robust voice interfaces and all you have to do is plug in your control logic — well, that is with **askhome**.

CHAPTER 3

User Guide

Quick Start

In here you'll find a fast introduction to askhome and show you how to get a simple Smart Home Skill up and running.

Installation

Install askhome with pip:

```
$ pip install askhome
```

If you are deploying to AWS Lambda by uploading zips, install askhome to your directory with:

```
$ pip install askhome -t /path/to/project-dir
```

More on *deployment* later.

Defining Appliances

When a Smart Home Skill is installed on Amazon Echo, Alexa first needs to discover all available appliances with information about what *actions* they support. With askhome, you can define your device types by subclassing `Appliance`. Methods marked with the `@Appliance.action` decorator will be discoverable and called when the corresponding request comes in:

```
from askhome import Appliance

class Light(Appliance):
    @Appliance.action
    def turn_on(self, request):
        pass # Your logic for switching a light on here

    @Appliance.action
    def turn_off(self, request):
        pass # Your logic for switching a light, you guessed it, off here
```

Now that we've defined a light appliance type, lets fill our smart home with some:

```
from askhome import Smarthome

home = Smarthome()
home.add_appliance('light1', Light, name='Kitchen Light',
                   description='Turn me on when cutting vegetables.')

home.add_appliance('light2', Light, name='Bedroom Light',
                   model='Very Bright Light 8000')
```

We've added two lights with some additional info. For all details you can set, check the `Smarthome.add_appliance` method or the official DiscoverAppliancesResponse documentation.

All that's left to do is expose a handler for our AWS Lambda instance (more on that in *deployment*):

```
lambda_handler = home.lambda_handler
```

And that's it! We've got a fully functioning Smart Home Skill that controls our lights. It can be initialized with:

> "Alexa, discover my devices."

And then to control the lights:

> "Alexa, turn on the kitchen light."

Handling Requests

When the method wrapped with `@Appliance.action` gets called, it receives a `Request` object as argument. It has some basic useful attributes such as `payload` or `name`, but what's special about it is that for every action type a specific `Request` subclass is passed. These subclasses have additional helpful attributes and a `response` method which simplifies the response creation. For instance a `set_target_temperature` action receives a request that can do this:

```
class Heater(Appliance):
    @Appliance.action
    def set_target_temperature(self, request):
        print request.temperature
        return request.response(request.temperature,
                                mode='HEAT',
                                previous_temperature=21.3,
                                previous_mode='AUTO')
```

If the action method doesn't return anything (returns `None`), success is implied.

Actions Overview

Possible action methods and their corresponding `Request` types passed are:

- turn_on(`Request`)
- turn_off(`Request`)
- set_percentage(`PercentageRequest`)

- increment_percentage(*PercentageRequest*)
- decrement_percentage(*PercentageRequest*)
- set_target_temperature(*ChangeTemperatureRequest*)
- increment_target_temperature(*ChangeTemperatureRequest*)
- decrement_target_temperature(*ChangeTemperatureRequest*)
- get_target_temperature(*GetTargetTemperatureRequest*)
- get_temperature_reading(*TemperatureReadingRequest*)
- set_lock_state(*LockStateRequest*)
- get_lock_state(*LockStateRequest*)

Here is a sample usage of all possible actions:

```
from askhome.requests import *

class UltimateAppliance(Appliance):

    # The action_for decorator can mark a method for multiple actions
    @Appliance.action_for('turn_on', 'turn_off')
    def turn_on_off(self, request):
        # type: (Request) -> Optional[dict]
        pass # nothing special here

    @Appliance.action_for('set_percentage', 'increment_percentage',
                          'decrement_percentage')
    def control_percentage(self, request):
        # type: (PercentageRequest) -> Optional[dict]
        print request.percentage
        print request.delta_percentage

    @Appliance.action_for('set_target_temperature',
                          'increment_target_temperature',
                          'decrement_target_temperature')
    def control_temperature(self, request):
        # type: (ChangeTemperatureRequest) -> Optional[dict]
        print request.temperature
        print request.delta_temperature
        return request.response(22.8,
                                mode='HEAT',
                                previous_temperature=21.3,
                                previous_mode='AUTO')

    @Appliance.action
    def get_target_temperature(self, request):
        # type: (GetTargetTemperatureRequest) -> Optional[dict]
        return request.response(21.8,
                                cooling_temperature=20
                                heating_temperature=23,
                                mode='CUSTOM',
                                mode_name='mode name')

    @Appliance.action
    def get_temperature_reading(self, request):
        # type: (TemperatureReadingRequest) -> Optional[dict]
        return request.response(21.8, timestamp=datetime.now())
    @Appliance.action_for('set_lock_state', 'get_lock_state')
    def lock_state(self, request):
        # type: (LockStateRequest) -> Optional[dict]
        return request.response('LOCKED')
```

For further information about these actions see the official documentation.

Error Responses

If the user asked an invalid request or something goes wrong during the action execution, the Smart Home API offers plenty of possible error responses. To respond with an error, simply raise one of askhome's exceptions, like this:

```
from askhome.exceptions import ValueOutOfRangeError

class Heater(Appliance):
    @Appliance.action
    def set_target_temperature(self, request):
        if request.temperature not in range(15, 25):
            raise ValueOutOfRangeError(15, 25)
```

All possible exceptions can be found *here* or at the official error messages documentation.

Deployment

Unlike the Custom Skills, Smart Home Skills have to be hosted on AWS Lambda instances. To create a skill and deploy it to Lambda, follow the official tutorial. When it comes to uploading your code, you have to package your libraries with it. You can do that with a local pip *installation* and then uploading a zip of your project with all its dependencies included.

Deploying with Zappa

Zappa is an awesome tool to deploy WSGI apps to Lambda. Smart Home Skills are not using the WSGI interface, but we can still use Zappa to automate our deployments. It also comes with advantages like precompiled python packages (such as pyOpenSSL) which would otherwise have to be compiled on AWS machines.

To use it, first create a virtualenv for your project:

```
$ virtualenv .venv
$ source .venv/Scripts/activate
```

Then install the required packages:

```
$ pip install Zappa askhome
```

Create a `zappa_settings.yml` configuration file for Zappa:

```
dev:
  s3_bucket: smart-home-skill-dev-deploy
  lambda_handler: main.lambda_handler # name of your file and exposed handler
  aws_region: us-east-1 # region has to match your Echo version
  timeout_seconds: 10
  memory_size: 128
  keep_warm: false
  touch: false # keep Zappa from sending WSGI requests to your skill
```

Finally, let Zappa do its work:

```
$ zappa deploy
```

That should create a Lambda function, but you still need to manually add the trigger and link the function to your skill as described in the official tutorial. After that your Echo should respond to your commands!

Next, you can go to the official Smart Home Skill API documentation for detailed request information or continue to *Advanced Usage*.

Advanced Usage

This section covers some more advanced features of askhome.

Reusing Appliance Details

If you're creating a Smart Home Skill for devices from one manufacturer, you probably don't want to repeat yourself when specifying the details when adding the device with `Smarthome.add_appliance`. You can set defaults to either `Appliance` or all devices in the `Smarthome` like this:

```
class Door(Appliance):
    class Details:
        model = 'QualityDoor'
        version = '2.0'

home = Smarthome(manufacturer='EvilCorp')
home.add_appliance('door1', Door, name='Front Door')
home.add_appliance('door2', Door, name='Back Door', version='1.0')
```

Resulting detail value is resolved in this order:

1. `Smarthome.add_appliance` keyword arguments
2. `Appliance.Details` inner class
3. `Smarthome.__init__` keyword arguments

Smarthome Handlers

The simple skill described in *Quick Start* has a potential problem in that it needs to know all appliances user has on every request. For example, if you keep the data in a remote database, it's wasteful to query for every appliance when switching one light on.

This problem can be solved by handling discovery and getting appliances for requests manually:

```
home = Smarthome()

@home.discover_handler
def discover(request):
    # Query the database here for all available appliances
    home.add_appliance('light1', Light, name='Kitchen Light',
                       additional_details={'type': 'Light'})
    return request.response(home)

@home.get_appliance_handler
def get_appliance(request):
    if request.appliance_details['type'] == 'Light':
        return Light
```

Here we've used the additional_details field the Smart Home API offers. You can save custom data in there during discovery and for every subsequent request you get that data back. This way, we query the database only once during discovery.

User Data

Often you will still need to query for some information about the user that sent the request. For that, there is another `Smarthome` decorator:

```
@home.prepare_handler
def prepare(request):
    # Query the database for ip address of the user
    ip = '1.2.3.4'
    request.custom_data = {'user_ip': ip}
```

The above `prepare` function gets called before every request is processed. We save our user data to `Request.custom_data` attribute, which we can use in any of our action methods.

CHAPTER 4

API Documentation

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

API Reference

Appliance class

class `askhome.`**`Appliance`**(*request=None*)

Appliance subclasses are used to describe what actions devices support.

Methods of subclasses can be marked with decorators (like `@Appliance.action`) and are used to generate the Alexa DiscoverApplianceResponse. Alexa control and query requests are then routed to the corresponding decorated method.

Appliance subclass can also contain a `Details` inner class for instance defaults during discovery (see `Smarthome.add_appliance` for possible attributes).

`request`

Request – Currently processed request.

`id`

str – Identifier of the appliance from the appliance.applianceId of request payload.

`additional_details`

dict – Information that was sent for the DiscoverAppliancesRequest. Some instance specific details can be saved here.

`__init__`(*request=None*)

Appliance gets initialized just before its action methods are called. Put your logic for preparation before handling the request here.

classmethod **`action`**(*func*)

Decorator for marking the method as an action sent for the DiscoverAppliancesRequest.

The action name is generated from the camelCased method name (e.g. turn_on -> turnOn). The decorated method should take request as an argument, specific subclass of `Request` is passed for each action.

Possible action methods and their corresponding `Request` types passed are:

- turn_on(*askhome.requests.Request*)
- turn_off(*askhome.requests.Request*)
- set_percentage(*askhome.requests.PercentageRequest*)
- increment_percentage(*askhome.requests.PercentageRequest*)
- decrement_percentage(*askhome.requests.PercentageRequest*)
- set_target_temperature(*askhome.requests.ChangeTemperatureRequest*)
- increment_target_temperature(*askhome.requests.ChangeTemperatureRequest*)
- decrement_target_temperature(*askhome.requests.ChangeTemperatureRequest*)
- get_target_temperature(*askhome.requests.GetTargetTemperatureRequest*)
- get_temperature_reading(*askhome.requests.TemperatureReadingRequest*)
- set_lock_state(*askhome.requests.LockStateRequest*)
- get_lock_state(*askhome.requests.LockStateRequest*)

classmethod `action_for`(**args*)
Decorator similar to the `action` decorator, except it doesn't generate the action name from the method name. All action names that should lead to the decorated method are passed as arguments to the decorator.

`actions`
dict(str, function) – All actions the appliance supports and their corresponding (unbound) method references. Action names are formatted for the DiscoverAppliancesRequest.

`request_handlers`
dict(str, function) – All requests the appliance supports (methods marked as actions) and their corresponding (unbound) method references. For example action turn_on would be formatted as TurnOnRequest.

class `Details`
Inner class in `Appliance` subclasses provides default values so that they don't have to be repeated in `Smarthome.add_appliance`.

Smarthome class

class `askhome.`**`Smarthome`**(***details*)
Holds information about all appliances and handles routing requests to appliance actions.

`appliances`
dict(str, (Appliance, dict)) – All registered appliances with details dict.

`details`
dict – Defaults for details of appliances during DiscoverAppliancesRequest.

`__init__`(***details*)

Parameters **`details`** (*`dict`*) – Defaults for details of appliances during DiscoverAppliancesRequest. See `add_appliance` method for possible values.

`add_appliance`(*appl_id*, *appl_class*, *name=None*, *description=None*, *additional_details=None*, *model=None*, *version=None*, *manufacturer=None*, *reachable=None*)
Register `Appliance` so it can be discovered and routed to.

The keyword arguments can be also defined in `Smarthome.__init__` and `Details` inner class in the appliance. Resulting value is resolved in order of priority: `Smarthome.add_appliance` kwargs -> `Appliance.Details` -> `Smarthome.__init__` kwargs

Parameters

- **`appl_id`** (*`str`*) – Unique identifier of the appliance, needs to be consistent across multiple discovery requests for the same device. Can contain any letter or number and the following special characters: _ - = # ; : ? @ &. Cannot exceed 256 characters.
- **`appl_class`** (`Appliance`) – `Appliance` subclass with marked actions.
- **`name`** (*`str`*) – Friendly name used by the customer to identify the device. Cannot exceed 128 characters and should not contain special characters or punctuation.
- **`description`** (*`str`*) – Human-readable description of the device. This value cannot exceed 128 characters. The description should contain a description of how the device is connected. For example, "WiFi Thermostat connected via Wink".
- **`additional_details`** (*`dict(str, str)`*) – Some instance specific details can be saved here. This field is sent back every time a request on that appliance is made. Cannot exceed 5000 bytes.
- **`model`** (*`str`*) – Device model name. Cannot exceed 128 characters.
- **`version`** (*`str`*) – Vendor-provided version of the device. Cannot exceed 128 characters.
- **`manufacturer`** (*`str`*) – Name of device manufacturer. Cannot exceed 128 characters.
- **`reachable`** (*`bool`*) – Indicate if device is currently reachable.

`prepare_handler`(*func*)

Decorator for a function that gets called before every request. Useful to modify the request processed, for instance add data to `Request.custom_data`

`discover_handler`(*func*)

Decorator for a function that handles the DiscoverAppliancesRequest instead of the `Smarthome`. This can be useful for situations where querying the list of all devices is too expensive to be done every request. Should be used in conjunction with the `get_appliance_handler` decorator.

`get_appliance_handler`(*func*)

Decorator for a function that handles getting the `Appliance` subclass instead of the `Smarthome`. Should be used in conjunction with the `get_appliance_handler` decorator.

`healthcheck_handler`(*func*)

Decorator for a function that handles `HealthCheckRequest`. Behaves the same as a regular action method.

`lambda_handler`(*data, context=None*)

Main entry point for handling requests. Pass the AWS Lambda events here.

Requests

`askhome.requests.`**`create_request`**(*data, context=None*)

Create a specific `Request` subclass according to the request type.

Each `Request` subclass has specific properties to access request data more easily and differing `response` arguments for direct response creation.

class `askhome.requests.`**`Request`**(*data, context=None*)

Base Request class for parsing Alexa request data.

data
dict – Raw event data from the lambda handler.

context
object – Context object from the lambda handler.

header
dict – Header of the Alexa request.

payload
dict – Payload of the Alexa request.

name
str – Request name from the `name` field in header.

access_token
str – OAuth token from the `accessToken` field in payload.

custom_data
Any – Attribute for saving custom data through `Smarthome.prepare_handler`

__init__(*data*, *context=None*)

appliance_id
str – Identifier of the appliance from the appliance.applianceId of request payload.

appliance_details
dict – Information that was sent for the DiscoverApplianceRequest in field `appliance.additionalApplianceDetails`

response_header(*name=None*)
Generate response header with copied values from the request and correct name.

raw_response(*payload=None*, *header=None*)
Compose response from raw payload and header dicts

response(**args*, ***kwargs*)
Return response with empty payload. Arguments and implementation of this method differ in each Request subclass.

exception_response(*exception*)
Create response from exception instance.

class `askhome.requests.`**DiscoverRequest**(*data*, *context=None*)
Request class for Alexa DiscoverAppliancesRequest.

response(*smarthome*)
Generate DiscoverAppliancesResponse from appliances added to the passed `Smarthome`.

Details of each appliance are resolved in order of priority: `Smarthome.add_appliance` kwargs -> `Appliance.Details` -> `Smarthome.__init__` kwargs

class `askhome.requests.`**PercentageRequest**(*data*, *context=None*)
Request class for Alexa Increment/Decrement/SetPercentageRequest.

percentage

delta_percentage

class `askhome.requests.`**ChangeTemperatureRequest**(*data*, *context=None*)
Request class for Alexa Increment/Decrement/SetTargetTemperatureRequest.

temperature

delta_temperature

response (*temperature*, *mode=None*, *previous_temperature=None*, *previous_mode=None*)

Parameters

- **temperature** (*float*) – Target temperature set by the device, in degrees Celsius.
- **mode** (*str*) – Temperature mode of device. Can be 'AUTO', 'COOL' or 'HEAT'.
- **previous_temperature** (*float*) – Previous target temperature in degrees Celsius.
- **previous_mode** (*str*) – Previous temperature mode.

class askhome.requests.**GetTargetTemperatureRequest** (*data*, *context=None*)
Request class for Alexa GetTargetTemperatureRequest.

response (*temperature=None*, *cooling_temperature=None*, *heating_temperature=None*, *mode='AUTO'*, *mode_name=None*, *timestamp=None*)

Parameters

- **temperature** (*float*) – Target temperature set by the device, in degrees Celsius.
- **cooling_temperature** (*float*) – Target temperature (setpoint) for cooling, in degrees Celsius, when a device has dual setpoints. Usually combined with heatingTargetTemperature.
- **heating_temperature** (*float*) – Target temperature (setpoint) for heating, in degrees Celsius, when a device has dual setpoints. Usually combined with coolingTargetTemperature.
- **mode** (*str*) – Temperature mode of device. Can be one of 'AUTO', 'COOL', 'HEAT', 'ECO', 'OFF', 'CUSTOM'.
- **mode_name** (*str*) – Friendly name of the mode when it differs from the canonical name. Required when mode is 'CUSTOM'.
- **timestamp** (*datetime | str*) – Time when the information was last retrieved.

class askhome.requests.**TemperatureReadingRequest** (*data*, *context=None*)
Request class for Alexa GetTemperatureReadingRequest.

response (*temperature*, *timestamp=None*)

Parameters

- **temperature** (*float*) – Current temperature reading, in degrees Celsius.
- **timestamp** (*datetime | str*) – Time when the information was last retrieved.

class askhome.requests.**LockStateRequest** (*data*, *context=None*)
Request class for Alexa Get/SetLockStateRequest.

lock_state

response (*lock_state*, *timestamp=None*)

Parameters

- **lock_state** (*str*) – Can be 'LOCKED' or 'UNLOCKED' for GetLockStateRequest, can be only 'LOCKED' for SetLockStateRequest (for security reasons).
- **timestamp** (*datetime | str*) – Time when the information was last retrieved.

class askhome.requests.**HealthCheckRequest** (*data*, *context=None*)
Request class for Alexa HealthCheckRequest.

response (*healthy*, *description*)

Exceptions

exception `askhome.exceptions.`**`AskhomeException`**(**args*, ***kwargs*)

Base askhome exception from which all inherit.

These exceptions can be raised in `Appliance` actions or manually passed to `Request.exception_response` to create an error response.

`namespace` **= 'Alexa.ConnectedHome.Control'**

`__init__`(**args*, ***kwargs*)

Parameters

- **`name`** (*`str`*) – Custom error name in header of generated response
- **`payload`** (*`dict`*) – Custom payload of generated response

exception `askhome.exceptions.`**`ValueOutOfRangeError`**(*min_val*, *max_val*, **args*, ***kwargs*)

Amazon docs: Indicates a customer request would set a target value to a value out of its supported range. For example, a customer asks, "Alexa, set the kitchen to 1000 degrees".

`__init__`(*min_val*, *max_val*, **args*, ***kwargs*)

exception `askhome.exceptions.`**`TargetOfflineError`**(**args*, ***kwargs*)

Amazon docs: Indicates that the target device is not connected to the customer's device cloud or is not on.

exception `askhome.exceptions.`**`NoSuchTargetError`**(**args*, ***kwargs*)

Amazon docs: Indicates that the target device cannot be found, meaning it was never configured by the end-user.

exception `askhome.exceptions.`**`BridgeOfflineError`**(**args*, ***kwargs*)

Amazon docs: Indicates the target device is connected to a home automation hub or bridge, which is powered off.

exception `askhome.exceptions.`**`DriverInternalError`**(**args*, ***kwargs*)

Amazon docs: Indicates a generic runtime error within the skill adapter. When possible, a more specific error should be returned.

exception `askhome.exceptions.`**`DependentServiceUnavailableError`**(*service_name*, **args*, ***kwargs*)

Amazon docs: Indicates that a skill adapter dependency is unavailable and the skill adapter cannot complete the request.

`__init__`(*service_name*, **args*, ***kwargs*)

exception `askhome.exceptions.`**`TargetConnectivityUnstableError`**(**args*, ***kwargs*)

Amazon docs: Indicates the cloud-connectivity for the target device is not stable and reliable.

exception `askhome.exceptions.`**`TargetBridgeConnectivityUnstableError`**(**args*, ***kwargs*)

Amazon docs: Indicates that cloud-connectivity for a home automation hub or bridge that connects the target device is unstable and unreliable.

exception `askhome.exceptions.`**`TargetFirmwareOutdatedError`**(*min_version*, *cur_version*, **args*, ***kwargs*)

Amazon docs: Indicates that the target device has outdated firmware.

`__init__`(*min_version*, *cur_version*, **args*, ***kwargs*)

exception `askhome.exceptions.`**`TargetBridgeFirmwareOutdatedError`**(*min_version*, *cur_version*, **args*, ***kwargs*)

Amazon docs: Indicates that the home automation hub or bridge that connects the target device has outdated firmware.

`__init__`(*min_version*, *cur_version*, **args*, ***kwargs*)

exception `askhome.exceptions.`**`TargetHardwareMalfunctionError`**(**args*, ***kwargs*)

Amazon docs: Indicates that the target device experienced a hardware malfunction.

exception `askhome.exceptions.`**`TargetBridgeHardwareMalfunctionError`**(**args*, ***kwargs*)

Amazon docs: Indicates that the home automation hub or bridge connecting the target device experienced a hardware malfunction

exception `askhome.exceptions.`**`UnableToGetValueError`**(*error_code*, *error_description=None*, **args*, ***kwargs*)

Amazon docs: Indicates that an error occurred while trying to get the specified value on the target device. When returning this error, an appropriate error_code value enables Alexa to respond appropriately for different kinds of failures. You only need to generate an error code appropriate for the target device.

`namespace` = **'Alexa.ConnectedHome.Query'**

`__init__`(*error_code*, *error_description=None*, **args*, ***kwargs*)

Parameters

- **`error_code`** (*`str`*) – Possible error codes are:
  - DEVICE_AJAR: Cannot get the specified state because the door is open.
  - DEVICE_BUSY: The device is busy
  - DEVICE_JAMMED: The device is jammed.
  - DEVICE_OVERHEATED: The device has overheated.
  - HARDWARE_FAILURE: Request failed because of an undetermined hardware failure.
  - LOW_BATTERY: The device's battery is low
  - NOT_CALIBRATED: The device is not calibrated.
- **`error_description`** (*`str`*) – non-required custom description

exception `askhome.exceptions.`**`UnableToSetValueError`**(*error_code*, *error_description=None*, **args*, ***kwargs*)

Amazon docs: Indicates that an error occurred while trying to set the specified value on the target device. When returning this error, an appropriate error_code value enables Alexa to respond appropriately for different kinds of failures. You only need to generate error codes appropriate for the target device.

`__init__`(*error_code*, *error_description=None*, **args*, ***kwargs*)

Parameters

- **`error_code`** (*`str`*) – Possible error codes are:
  - DEVICE_AJAR: Cannot get the specified state because the door is open.
  - DEVICE_BUSY: The device is busy
  - DEVICE_JAMMED: The device is jammed.
  - DEVICE_OVERHEATED: The device has overheated.
  - HARDWARE_FAILURE: Request failed because of an undetermined hardware failure.
  - LOW_BATTERY: The device's battery is low
  - NOT_CALIBRATED: The device is not calibrated.
- **`error_description`** (*`str`*) – non-required custom description

exception `askhome.exceptions.`**`UnwillingToSetValueError`**(*error_code='ThermostatIsOff'*, *error_description=None*, **args*, ***kwargs*)
Amazon docs: Indicates that the target device partner is unwilling to set the requested value on the specified device. Use this error for temperature settings.

`__init__`(*error_code='ThermostatIsOff'*, *error_description=None*, **args*, ***kwargs*)

exception `askhome.exceptions.`**`RateLimitExceededError`**(*rate_limit*, *time_unit='HOUR'*, **args*, ***kwargs*)
Amazon docs: Indicates that the maximum number of requests that a device accepts has been exceeded. This message provides information about the maximum number of requests for a device and the time unit for those requests. For example, if a device accepts four requests per hour, the message should specify 4 and HOUR as rate_limit and time_unit, respectively.

`__init__`(*rate_limit*, *time_unit='HOUR'*, **args*, ***kwargs*)

exception `askhome.exceptions.`**`NotSupportedInCurrentModeError`**(*current_mode*, **args*, ***kwargs*)
Amazon docs: Indicates that the target device is in a mode in which it cannot be controlled with the Smart Home Skill API, and provides information about the current mode of the device.

`__init__`(*current_mode*, **args*, ***kwargs*)

exception `askhome.exceptions.`**`ExpiredAccessTokenError`**(**args*, ***kwargs*)
Amazon docs: Indicates that the access token used for authentication has expired and is no longer valid.

exception `askhome.exceptions.`**`InvalidAccessTokenError`**(**args*, ***kwargs*)
Amazon docs: Indicates that the access token used for authentication is not valid for a reason other than it has expired.

exception `askhome.exceptions.`**`UnsupportedTargetError`**(**args*, ***kwargs*)
Amazon docs: Indicates that the target device is not supported by the skill adapter.

exception `askhome.exceptions.`**`UnsupportedOperationError`**(**args*, ***kwargs*)
Amazon docs: Indicates that the requested operation is not supported on the target device.

exception `askhome.exceptions.`**`UnsupportedTargetSettingError`**(**args*, ***kwargs*)
Amazon docs: Indicates that the requested setting is not valid for the specified device and operation.

exception `askhome.exceptions.`**`UnexpectedInformationReceivedError`**(*faulting_parameter*, **args*, ***kwargs*)
Amazon docs: The request message or payload could not be handled by the skill adapter because it was malformed.

`__init__`(*faulting_parameter*, **args*, ***kwargs*)

Utils

`askhome.utils.`**`get_action_string`**(*func_name*)
Transform function name to Alexa action

`askhome.utils.`**`get_request_string`**(*func_name*)
Transform function name to Alexa request name

`askhome.utils.`**`rstrip_word`**(*text*, *suffix*)
Strip suffix from end of text

Python Module Index

Index