aws / **chalice**

Python Serverless Microframework for AWS

#python #aws #aws-lambda #cloud #serverless #serverless-framework #aws-apigateway #lambda #python3 #python27

1,379 commits | **11** branches | **33** releases | **72** contributors | Apache-2.0

Branch: master ▾ | New pull request | Create new file | Upload files | Find file | Clone or download ▾

joguSD Merge pull request #1014 from joguSD/changelogs ··· Latest commit d76dcc8 3 days ago

.github	Merge branch 'hyandell-master'	7 months ago
chalice	Merge pull request #664 from nplutt/logging-errors	3 days ago
docs	Fix documentation to correct custom authorizer	11 days ago
scripts	Set patch version to 0 on minor version bumps	a year ago
tests	Merge pull request #664 from nplutt/logging-errors	3 days ago
.coveragerc	Ignore not implementederror in coverage	2 years ago
.gitignore	Add Python information to chalice --version	6 months ago
.pylintrc	Ignore relative-import error to get pylint working on py2	5 days ago
.travis.yml	Add support for python 3.7	5 days ago
CHANGELOG.rst	Add changelog entries for new enhancements	3 days ago
CODE_OF_CONDUCT.rst	Convert code of conduct to rst	7 months ago
CONTRIBUTING.rst	Fix doc linting issues	a year ago
LICENSE	Initial commit of chalice, a serverless microframework	3 years ago
MANIFEST.in	Include policies json in manifest	3 years ago
Makefile	Fix flake8 warnings by using latest version	5 days ago
NOTICE	Initial commit of chalice, a serverless microframework	3 years ago
README.rst	Support bytes for the application/json binary type	4 days ago
requirements-dev.txt	Fix pylint version env marker for py2	5 days ago
requirements-docs.txt	Use guzzle_sphinx_theme	2 years ago
setup.cfg	Initial commit of chalice, a serverless microframework	3 years ago
setup.py	Bumping version to 1.6.1	a month ago

README.rst

Python Serverless Microframework for AWS

chat on gitter | build passing | docs passing | codecov 95%

Chalice is a microframework for writing serverless apps in python. It allows you to quickly create and deploy applications that use AWS Lambda. It provides:

- A command line tool for creating, deploying, and managing your app
- A decorator based API for integrating with Amazon API Gateway, Amazon S3, Amazon SNS, Amazon SQS, and other AWS services.
- Automatic IAM policy generation

You can create Rest APIs:

```
from chalice import Chalice

app = Chalice(app_name="helloworld")
```

```
@app.route("/")
def index():
    return {"hello": "world"}
```

Tasks that run on a periodic basis:

```
from chalice import Chalice, Rate

app = Chalice(app_name="helloworld")

# Automatically runs every 5 minutes
@app.schedule(Rate(5, unit=Rate.MINUTES))
def periodic_task(event):
    return {"hello": "world"}
```

You can connect a lambda function to an S3 event:

```
from chalice import Chalice

app = Chalice(app_name="helloworld")

# Whenever an object is uploaded to 'mybucket'
# this lambda function will be invoked.

@app.on_s3_event(bucket='mybucket')
def handler(event):
    print("Object uploaded for bucket: %s, key: %s"
          % (event.bucket, event.key))
```

As well as an SQS queue:

```
from chalice import Chalice

app = Chalice(app_name="helloworld")

# Invoke this lambda function whenever a message
# is sent to the ``my-queue-name`` SQS queue.

@app.on_sqs_message(queue='my-queue-name')
def handler(event):
    for record in event:
        print("Message body: %s" % record.body)
```

And several other AWS resources.

Once you've written your code, you just run `chalice deploy` and Chalice takes care of deploying your app.

```
$ chalice deploy
...
https://endpoint/dev

$ curl https://endpoint/api
{"hello": "world"}
```

Up and running in less than 30 seconds. Give this project a try and share your feedback with us here on Github.

The documentation is available on readthedocs.

Quickstart

In this tutorial, you'll use the `chalice` command line utility to create and deploy a basic REST API. First, you'll need to install `chalice`. Using a virtualenv is recommended:

```
$ pip install virtualenv
$ virtualenv ~/.virtualenvs/chalice-demo
$ source ~/.virtualenvs/chalice-demo/bin/activate
```

Note: **make sure you are using python2.7 or python3.6**. The `chalice` CLI as well as the `chalice` python package will support the versions of python supported by AWS Lambda. Currently, AWS Lambda supports python2.7 and python3.6, so that's what this project supports. You can ensure you're creating a virtualenv with python3.6 by running:

```
# Double check you have python3.6
$ which python3.6
/usr/local/bin/python3.6
$ virtualenv --python $(which python3.6) ~/.virtualenvs/chalice-demo
$ source ~/.virtualenvs/chalice-demo/bin/activate
```

Next, in your virtualenv, install `chalice` :

```
$ pip install chalice
```

You can verify you have chalice installed by running:

```
$ chalice --help
Usage: chalice [OPTIONS] COMMAND [ARGS]...
...
```

Credentials

Before you can deploy an application, be sure you have credentials configured. If you have previously configured your machine to run boto3 (the AWS SDK for Python) or the AWS CLI then you can skip this section.

If this is your first time configuring credentials for AWS you can follow these steps to quickly get started:

```
$ mkdir ~/.aws
$ cat >> ~/.aws/config
[default]
aws_access_key_id=YOUR_ACCESS_KEY_HERE
aws_secret_access_key=YOUR_SECRET_ACCESS_KEY
region=YOUR_REGION (such as us-west-2, us-west-1, etc)
```

If you want more information on all the supported methods for configuring credentials, see the boto3 docs.

Creating Your Project

The next thing we'll do is use the `chalice` command to create a new project:

```
$ chalice new-project helloworld
```

This will create a `helloworld` directory. Cd into this directory. You'll see several files have been created for you:

```
$ cd helloworld
$ ls -la
drwxr-xr-x   .chalice
-rw-r--r--   app.py
-rw-r--r--   requirements.txt
```

You can ignore the `.chalice` directory for now, the two main files we'll focus on is `app.py` and `requirements.txt` .

Let's take a look at the `app.py` file:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/')
def index():
    return {'hello': 'world'}
```

The `new-project` command created a sample app that defines a single view, `/`, that when called will return the JSON body `{"hello": "world"}`.

Deploying

Let's deploy this app. Make sure you're in the `helloworld` directory and run `chalice deploy`:

```
$ chalice deploy
...
Initiating first time deployment...
https://qxea58oupc.execute-api.us-west-2.amazonaws.com/api/
```

You now have an API up and running using API Gateway and Lambda:

```
$ curl https://qxea58oupc.execute-api.us-west-2.amazonaws.com/api/
{"hello": "world"}
```

Try making a change to the returned dictionary from the `index()` function. You can then redeploy your changes by running `chalice deploy`.

For the rest of these tutorials, we'll be using `httpie` instead of `curl` (https://github.com/jakubroztocil/httpie) to test our API. You can install `httpie` using `pip install httpie`, or if you're on Mac, you can run `brew install httpie`. The Github link has more information on installation instructions. Here's an example of using `httpie` to request the root resource of the API we just created. Note that the command name is `http`:

```
$ http https://qxea58oupc.execute-api.us-west-2.amazonaws.com/api/
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 18
Content-Type: application/json
Date: Mon, 30 May 2016 17:55:50 GMT
X-Cache: Miss from cloudfront

{
    "hello": "world"
}
```

Additionally, the API Gateway endpoints will be shortened to `https://endpoint/api/` for brevity. Be sure to substitute `https://endpoint/api/` for the actual endpoint that the `chalice` CLI displays when you deploy your API (it will look something like `https://abcdefg.execute-api.us-west-2.amazonaws.com/api/`.

Next Steps

You've now created your first app using `chalice`.

The next few sections will build on this quickstart section and introduce you to additional features including: URL parameter capturing, error handling, advanced routing, current request metadata, and automatic policy generation.

Tutorial: URL Parameters

Now we're going to make a few changes to our `app.py` file that demonstrate additional capabilities provided by the python serverless microframework for AWS.

Our application so far has a single view that allows you to make an HTTP GET request to `/`. Now let's suppose we want to capture parts of the URI:

```python
from chalice import Chalice

app = Chalice(app_name='helloworld')

CITIES_TO_STATE = {
    'seattle': 'WA',
    'portland': 'OR',
}
```

```
@app.route('/')
def index():
    return {'hello': 'world'}

@app.route('/cities/{city}')
def state_of_city(city):
    return {'state': CITIES_TO_STATE[city]}
```

In the example above, we've now added a `state_of_city` view that allows a user to specify a city name. The view function takes the city name and returns name of the state the city is in. Notice that the `@app.route` decorator has a URL pattern of `/cities/{city}` . This means that the value of `{city}` is captured and passed to the view function. You can also see that the `state_of_city` takes a single argument. This argument is the name of the city provided by the user. For example:

```
GET /cities/seattle   --> state_of_city('seattle')
GET /cities/portland  --> state_of_city('portland')
```

Now that we've updated our `app.py` file with this new view function, let's redeploy our application. You can run `chalice deploy` from the `helloworld` directory and it will deploy your application:

```
$ chalice deploy
```

Let's try it out. Note the examples below use the `http` command from the `httpie` package. You can install this using `pip install httpie` :

```
$ http https://endpoint/api/cities/seattle
HTTP/1.1 200 OK

{
    "state": "WA"
}

$ http https://endpoint/api/cities/portland
HTTP/1.1 200 OK

{
    "state": "OR"
}
```

Notice what happens if we try to request a city that's not in our `CITIES_TO_STATE` map:

```
$ http https://endpoint/api/cities/vancouver
HTTP/1.1 500 Internal Server Error
Content-Type: application/json
X-Cache: Error from cloudfront

{
    "Code": "ChaliceViewError",
    "Message": "ChaliceViewError: An internal server error occurred."
}
```

In the next section, we'll see how to fix this and provide better error messages.

Tutorial: Error Messages

In the example above, you'll notice that when our app raised an uncaught exception, a 500 internal server error was returned.

In this section, we're going to show how you can debug and improve these error messages.

The first thing we're going to look at is how we can debug this issue. By default, debugging is turned off, but you can enable debugging to get more information:

```
from chalice import Chalice
```

```
app = Chalice(app_name='helloworld')
app.debug = True
```

The `app.debug = True` enables debugging for your app. Save this file and redeploy your changes:

```
$ chalice deploy
...
https://endpoint/api/
```

Now, when you request the same URL that returned an internal server error, you'll get back the original stack trace:

```
$ http https://endpoint/api/cities/vancouver
Traceback (most recent call last):
  File "/var/task/chalice/app.py", line 304, in _get_view_function_response
    response = view_function(*function_args)
  File "/var/task/app.py", line 18, in state_of_city
    return {'state': CITIES_TO_STATE[city]}
KeyError: u'vancouver'
```

We can see that the error is caused from an uncaught `KeyError` resulting from trying to access the `vancouver` key.

Now that we know the error, we can fix our code. What we'd like to do is catch this exception and instead return a more helpful error message to the user. Here's the updated code:

```
from chalice import BadRequestError

@app.route('/cities/{city}')
def state_of_city(city):
    try:
        return {'state': CITIES_TO_STATE[city]}
    except KeyError:
        raise BadRequestError("Unknown city '%s', valid choices are: %s" % (
            city, ', '.join(CITIES_TO_STATE.keys())))
```

Save and deploy these changes:

```
$ chalice deploy
$ http https://endpoint/api/cities/vancouver
HTTP/1.1 400 Bad Request

{
    "Code": "BadRequestError",
    "Message": "Unknown city 'vancouver', valid choices are: portland, seattle"
}
```

We can see now that we have received a `Code` and `Message` key, with the message being the value we passed to `BadRequestError`. Whenever you raise a `BadRequestError` from your view function, the framework will return an HTTP status code of 400 along with a JSON body with a `Code` and `Message`. There are a few additional exceptions you can raise from your python code:

```
* BadRequestError - return a status code of 400
* UnauthorizedError - return a status code of 401
* ForbiddenError - return a status code of 403
* NotFoundError - return a status code of 404
* ConflictError - return a status code of 409
* UnprocessableEntityError - return a status code of 422
* TooManyRequestsError - return a status code of 429
* ChaliceViewError - return a status code of 500
```

You can import these directly from the `chalice` package:

```
from chalice import UnauthorizedError
```

Tutorial: Additional Routing

So far, our examples have only allowed GET requests. It's actually possible to support additional HTTP methods. Here's an example of a view function that supports PUT:

```
@app.route('/resource/{value}', methods=['PUT'])
def put_test(value):
    return {"value": value}
```

We can test this method using the `http` command:

```
$ http PUT https://endpoint/api/resource/foo
HTTP/1.1 200 OK

{
    "value": "foo"
}
```

Note that the `methods` kwarg accepts a list of methods. Your view function will be called when any of the HTTP methods you specify are used for the specified resource. For example:

```
@app.route('/myview', methods=['POST', 'PUT'])
def myview():
    pass
```

The above view function will be called when either an HTTP POST or PUT is sent to `/myview` .

Alternatively if you do not want to share the same view function across multiple HTTP methods for the same route url, you may define separate view functions to the same route url but have the view functions differ by HTTP method. For example:

```
@app.route('/myview', methods=['POST'])
def myview_post():
    pass

@app.route('/myview', methods=['PUT'])
def myview_put():
    pass
```

This setup will route all HTTP POST's to `/myview` to the `myview_post()` view function and route all HTTP PUT's to `/myview` to the `myview_put()` view function. It is also important to note that the view functions **must** have unique names. For example, both view functions cannot be named `myview()` .

In the next section we'll go over how you can introspect the given request in order to differentiate between various HTTP methods.

Tutorial: Request Metadata

In the examples above, you saw how to create a view function that supports an HTTP PUT request as well as a view function that supports both POST and PUT via the same view function. However, there's more information we might need about a given request:

- In a PUT/POST, you frequently send a request body. We need some way of accessing the contents of the request body.
- For view functions that support multiple HTTP methods, we'd like to detect which HTTP method was used so we can have different code paths for PUTs vs. POSTs.

All of this and more is handled by the current request object that the `chalice` library makes available to each view function when it's called.

Let's see an example of this. Suppose we want to create a view function that allowed you to PUT data to an object and retrieve that data via a corresponding GET. We could accomplish that with the following view function:

```
from chalice import NotFoundError

OBJECTS = {
}
```

```
@app.route('/objects/{key}', methods=['GET', 'PUT'])
def myobject(key):
    request = app.current_request
    if request.method == 'PUT':
        OBJECTS[key] = request.json_body
    elif request.method == 'GET':
        try:
            return {key: OBJECTS[key]}
        except KeyError:
            raise NotFoundError(key)
```

Save this in your `app.py` file and rerun `chalice deploy`. Now, you can make a PUT request to `/objects/your-key` with a request body, and retrieve the value of that body by making a subsequent `GET` request to the same resource. Here's an example of its usage:

```
# First, trying to retrieve the key will return a 404.
$ http GET https://endpoint/api/objects/mykey
HTTP/1.1 404 Not Found

{
    "Code": "NotFoundError",
    "Message": "mykey"
}

# Next, we'll create that key by sending a PUT request.
$ echo '{"foo": "bar"}' | http PUT https://endpoint/api/objects/mykey
HTTP/1.1 200 OK

null

# And now we no longer get a 404, we instead get the value we previously
# put.
$ http GET https://endpoint/api/objects/mykey
HTTP/1.1 200 OK

{
    "mykey": {
        "foo": "bar"
    }
}
```

You might see a problem with storing the objects in a module level `OBJECTS` variable. We address this in the next section.

The `app.current_request` object also has the following properties.

- `current_request.query_params` - A dict of the query params for the request.
- `current_request.headers` - A dict of the request headers.
- `current_request.uri_params` - A dict of the captured URI params.
- `current_request.method` - The HTTP method (as a string).
- `current_request.json_body` - The parsed JSON body (`json.loads(raw_body)`)
- `current_request.raw_body` - The raw HTTP body as bytes.
- `current_request.context` - A dict of additional context information
- `current_request.stage_vars` - Configuration for the API Gateway stage

Don't worry about the `context` and `stage_vars` for now. We haven't discussed those concepts yet. The `current_request` object also has a `to_dict` method, which returns all the information about the current request as a dictionary. Let's use this method to write a view function that returns everything it knows about the request:

```
@app.route('/introspect')
def introspect():
    return app.current_request.to_dict()
```

Save this to your `app.py` file and redeploy with `chalice deploy`. Here's an example of hitting the `/introspect` URL. Note how we're sending a query string as well as a custom `X-TestHeader` header:

```
$ http 'https://endpoint/api/introspect?query1=value1&query2=value2' 'X-TestHeader: Foo'
HTTP/1.1 200 OK
```

```
{
    "context": {
        "apiId": "apiId",
        "httpMethod": "GET",
        "identity": {
            "accessKey": null,
            "accountId": null,
            "apiKey": null,
            "caller": null,
            "cognitoAuthenticationProvider": null,
            "cognitoAuthenticationType": null,
            "cognitoIdentityId": null,
            "cognitoIdentityPoolId": null,
            "sourceIp": "1.1.1.1",
            "userAgent": "HTTPie/0.9.3",
            "userArn": null
        },
        "requestId": "request-id",
        "resourceId": "resourceId",
        "resourcePath": "/introspect",
        "stage": "dev"
    },
    "headers": {
        "accept": "*/*",
        ...
        "x-testheader": "Foo"
    },
    "method": "GET",
    "query_params": {
        "query1": "value1",
        "query2": "value2"
    },
    "raw_body": null,
    "stage_vars": null,
    "uri_params": null
}
```

Tutorial: Request Content Types

The default behavior of a view function supports a request body of `application/json`. When a request is made with a `Content-Type` of `application/json`, the `app.current_request.json_body` attribute is automatically set for you. This value is the parsed JSON body.

You can also configure a view function to support other content types. You can do this by specifying the `content_types` parameter value to your `app.route` function. This parameter is a list of acceptable content types. Here's an example of this feature:

```python
import sys

from chalice import Chalice
if sys.version_info[0] == 3:
    # Python 3 imports.
    from urllib.parse import urlparse, parse_qs
else:
    # Python 2 imports.
    from urlparse import urlparse, parse_qs


app = Chalice(app_name='helloworld')


@app.route('/', methods=['POST'],
           content_types=['application/x-www-form-urlencoded'])
def index():
    parsed = parse_qs(app.current_request.raw_body.decode())
    return {
        'states': parsed.get('states', [])
    }
```

There's a few things worth noting in this view function. First, we've specified that we only accept the `application/x-www-form-urlencoded` content type. If we try to send a request with `application/json`, we'll now get a `415 Unsupported Media Type` response:

```
$ http POST https://endpoint/api/ states=WA states=CA --debug
...
>>> requests.request(**{'allow_redirects': False,
 'headers': {'Accept': 'application/json',
             'Content-Type': 'application/json',
...


HTTP/1.1 415 Unsupported Media Type

{
    "message": "Unsupported Media Type"
}
```

If we use the `--form` argument, we can see the expected behavior of this view function because `httpie` sets the `Content-Type` header to `application/x-www-form-urlencoded`:

```
$ http --form POST https://endpoint/api/formtest states=WA states=CA --debug
...
>>> requests.request(**{'allow_redirects': False,
 'headers': {'Content-Type': 'application/x-www-form-urlencoded; charset=utf-8',
...

HTTP/1.1 200 OK
{
    "states": [
        "WA",
        "CA"
    ]
}
```

The second thing worth noting is that `app.current_request.json_body` **is only available for the application/json content type.** In our example above, we used `app.current_request.raw_body` to access the raw body bytes:

```
parsed = parse_qs(app.current_request.raw_body)
```

`app.current_request.json_body` is set to `None` whenever the `Content-Type` is not `application/json`. This means that you will need to use `app.current_request.raw_body` and parse the request body as needed.

Tutorial: Customizing the HTTP Response

The return value from a chalice view function is serialized as JSON as the response body returned back to the caller. This makes it easy to create rest APIs that return JSON response bodies.

Chalice allows you to control this behavior by returning an instance of a chalice specific `Response` class. This behavior allows you to:

- Specify the status code to return
- Specify custom headers to add to the response
- Specify response bodies that are not `application/json`

Here's an example of this:

```
from chalice import Chalice, Response

app = Chalice(app_name='custom-response')


@app.route('/')
def index():
    return Response(body='hello world!',
                    status_code=200,
                    headers={'Content-Type': 'text/plain'})
```

This will result in a plain text response body:

```
$ http https://endpoint/api/
HTTP/1.1 200 OK
Content-Length: 12
Content-Type: text/plain

hello world!
```

Tutorial: GZIP compression for json

The return value from a chalice view function is serialized as JSON as the response body returned back to the caller. This makes it easy to create rest APIs that return JSON response bodies.

Chalice allows you to control this behavior by returning an instance of a chalice specific `Response` class. This behavior allows you to:

- Add `application/json` to binary_types
- Specify the status code to return
- Specify custom header `Content-Type: application/json`
- Specify custom header `Content-Encoding: gzip`

Here's an example of this:

```python
import json
import gzip
from chalice import Chalice, Response

app = Chalice(app_name='compress-response')
app.api.binary_types.append('application/json')

@app.route('/')
def index():
    blob = json.dumps({'hello': 'world'}).encode('utf-8')
    payload = gzip.compress(blob)
    custom_headers = {
        'Content-Type': 'application/json',
        'Content-Encoding': 'gzip'
    }
    return Response(body=payload,
                    status_code=200,
                    headers=custom_headers)
```

Tutorial: CORS Support

You can specify whether a view supports CORS by adding the `cors=True` parameter to your `@app.route()` call. By default this value is false:

```python
@app.route('/supports-cors', methods=['PUT'], cors=True)
def supports_cors():
    return {}
```

Settings `cors=True` has similar behavior to enabling CORS using the AWS Console. This includes:

- Injecting the `Access-Control-Allow-Origin: *` header to your responses, including all error responses you can return.
- Automatically adding an `OPTIONS` method to support preflighting requests.

The preflight request will return a response that includes:

- `Access-Control-Allow-Origin: *`
- The `Access-Control-Allow-Methods` header will return a list of all HTTP methods you've called out in your view function. In the example above, this will be `PUT,OPTIONS` .
- `Access-Control-Allow-Headers: Content-Type,X-Amz-Date,Authorization, X-Api-Key,X-Amz-Security-Token` .

If more fine grained control of the CORS headers is desired, set the `cors` parameter to an instance of `CORSConfig` instead of `True`. The `CORSConfig` object can be imported from from the `chalice` package it's constructor takes the following keyword arguments that map to CORS headers:

Argument	Type	Header
allow_origin	str	Access-Control-Allow-Origin
allow_headers	list	Access-Control-Allow-Headers
expose_headers	list	Access-Control-Expose-Headers
max_age	int	Access-Control-Max-Age
allow_credentials	bool	Access-Control-Allow-Credentials

Code sample defining more CORS headers:

```
from chalice import CORSConfig
cors_config = CORSConfig(
    allow_origin='https://foo.example.com',
    allow_headers=['X-Special-Header'],
    max_age=600,
    expose_headers=['X-Special-Header'],
    allow_credentials=True
)
@app.route('/custom_cors', methods=['GET'], cors=cors_config)
def supports_custom_cors():
    return {'cors': True}
```

There's a couple of things to keep in mind when enabling cors for a view:

- An `OPTIONS` method for preflighting is always injected. Ensure that you don't have `OPTIONS` in the `methods=[...]` list of your view function.
- Even though the `Access-Control-Allow-Origin` header can be set to a string that is a space separated list of origins, this behavior does not work on all clients that implement CORS. You should only supply a single origin to the `CORSConfig` object. If you need to supply multiple origins you will need to define a custom handler for it that accepts `OPTIONS` requests and matches the `Origin` header against a whitelist of origins. If the match is successful then return just their `Origin` back to them in the `Access-Control-Allow-Origin` header.

  Example:

```
from chalice import Chalice, Response

app = Chalice(app_name='multipleorigincors')

_ALLOWED_ORIGINS = set([
    'http://allowed1.example.com',
    'http://allowed2.example.com',
])


@app.route('/cors_multiple_origins', methods=['GET', 'OPTIONS'])
def supports_cors_multiple_origins():
    method = app.current_request.method
    if method == 'OPTIONS':
        headers = {
            'Access-Control-Allow-Method': 'GET,OPTIONS',
            'Access-Control-Allow-Origin': ','.join(_ALLOWED_ORIGINS),
            'Access-Control-Allow-Headers': 'X-Some-Header',
        }
        origin = app.current_request.headers.get('origin', '')
        if origin in _ALLOWED_ORIGINS:
            headers.update({'Access-Control-Allow-Origin': origin})
        return Response(
            body=None,
            headers=headers,
        )
    elif method == 'GET':
        return 'Foo'
```

- Every view function must explicitly enable CORS support.

The last point will change in the future. See this issue for more information.

Tutorial: Policy Generation

In the previous section we created a basic rest API that allowed you to store JSON objects by sending the JSON in the body of an HTTP PUT request to `/objects/{name}` . You could then retrieve objects by sending a GET request to `/objects/{name}` .

However, there's a problem with the code we wrote:

```
OBJECTS = {
}

@app.route('/objects/{key}', methods=['GET', 'PUT'])
def myobject(key):
    request = app.current_request
    if request.method == 'PUT':
        OBJECTS[key] = request.json_body
    elif request.method == 'GET':
        try:
            return {key: OBJECTS[key]}
        except KeyError:
            raise NotFoundError(key)
```

We're storing the key value pairs in a module level `OBJECTS` variable. We can't rely on local storage like this persisting across requests.

A better solution would be to store this information in Amazon S3. To do this, we're going to use boto3, the AWS SDK for Python. First, install boto3:

```
$ pip install boto3
```

Next, add `boto3` to your requirements.txt file:

```
$ echo 'boto3==1.3.1' >> requirements.txt
```

The requirements.txt file should be in the same directory that contains your `app.py` file. Next, let's update our view code to use boto3:

```
import json
import boto3
from botocore.exceptions import ClientError

from chalice import NotFoundError


S3 = boto3.client('s3', region_name='us-west-2')
BUCKET = 'your-bucket-name'


@app.route('/objects/{key}', methods=['GET', 'PUT'])
def s3objects(key):
    request = app.current_request
    if request.method == 'PUT':
        S3.put_object(Bucket=BUCKET, Key=key,
                      Body=json.dumps(request.json_body))
    elif request.method == 'GET':
        try:
            response = S3.get_object(Bucket=BUCKET, Key=key)
            return json.loads(response['Body'].read())
        except ClientError as e:
            raise NotFoundError(key)
```

Make sure to change `BUCKET` with the name of an S3 bucket you own. Redeploy your changes with `chalice deploy`. Now, whenever we make a `PUT` request to `/objects/keyname`, the data send will be stored in S3. Any subsequent `GET` requests will retrieve this data from S3.

Manually Providing Policies

IAM permissions can be auto generated, provided manually or can be pre-created and explicitly configured. To use a pre-configured IAM role ARN for chalice, add these two keys to your chalice configuration. Setting manage_iam_role to false tells Chalice to not attempt to generate policies and create IAM role.

```
"manage_iam_role":false
"iam_role_arn":"arn:aws:iam::<account-id>:role/<role-name>"
```

Whenever your application is deployed using `chalice`, the auto generated policy is written to disk at `<projectdir>/.chalice/policy.json`. When you run the `chalice deploy` command, you can also specify the `--no-autogen-policy` option. Doing so will result in the `chalice` CLI loading the `<projectdir>/.chalice/policy.json` file and using that file as the policy for the IAM role. You can manually edit this file and specify `--no-autogen-policy` if you'd like to have full control over what IAM policy to associate with the IAM role.

You can also run the `chalice gen-policy` command from your project directory to print the auto generated policy to stdout. You can then use this as a starting point for your policy.

```
$ chalice gen-policy
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:ListAllMyBuckets"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow",
      "Sid": "9155de6ad1d74e4c8b1448255770e60c"
    }
  ]
}
```

Experimental Status

The automatic policy generation is still in the early stages, it should be considered experimental. You can always disable policy generation with `--no-autogen-policy` for complete control.

Additionally, you will be prompted for confirmation whenever the auto policy generator detects actions that it would like to add or remove:

```
$ chalice deploy
Updating IAM policy.

The following action will be added to the execution policy:

s3:ListBucket

Would you like to continue?  [Y/n]:
```

Tutorial: Using Custom Authentication

AWS API Gateway routes can be authenticated in multiple ways:

- API Key
- AWS IAM
- Cognito User Pools
- Custom Auth Handler

API Key

```
@app.route('/authenticated', methods=['GET'], api_key_required=True)
def authenticated():
    return {"secure": True}
```

Only requests sent with a valid X-Api-Key header will be accepted.

Using AWS IAM

```
authorizer = IAMAuthorizer()

@app.route('/iam-role', methods=['GET'], authorizer=authorizer)
def authenticated():
    return {"secure": True}
```

Using Amazon Cognito User Pools

To integrate with cognito user pools, you can use the `CognitoUserPoolAuthorizer` object:

```
authorizer = CognitoUserPoolAuthorizer(
    'MyPool', header='Authorization',
    provider_arns=['arn:aws:cognito:...:userpool/name'])

@app.route('/user-pools', methods=['GET'], authorizer=authorizer)
def authenticated():
    return {"secure": True}
```

Note, earlier versions of chalice also have an `app.define_authorizer` method as well as an `authorizer_name` argument on the `@app.route(...)` method. This approach is deprecated in favor of `CognitoUserPoolAuthorizer` and the `authorizer` argument in the `@app.route(...)` method. `app.define_authorizer` will be removed in future versions of chalice.

Using Custom Authorizers

To integrate with custom authorizers, you can use the `CustomAuthorizer` method on the `app` object. You'll need to set the `authorizer_uri` to the URI of your lambda function.

```
authorizer = CustomAuthorizer(
    'MyCustomAuth', header='Authorization',
    authorizer_uri=('arn:aws:apigateway:region:lambda:path/2015-03-31'
                    '/functions/arn:aws:lambda:region:account-id:'
                    'function:FunctionName/invocations'))

@app.route('/custom-auth', methods=['GET'], authorizer=authorizer)
def authenticated():
    return {"secure": True}
```

Tutorial: Local Mode

As you develop your application, you may want to experiment locally before deploying your changes. You can use `chalice local` to spin up a local HTTP server you can use for testing.

For example, if we have the following `app.py` file:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/')
def index():
    return {'hello': 'world'}
```

We can run `chalice local` to test this API locally:

> $ chalice local Serving on localhost:8000

We can override the port using:

> $ chalice local --port=8080

We can now test our API using `localhost:8000` :

```
$ http localhost:8000/
HTTP/1.0 200 OK
Content-Length: 18
Content-Type: application/json
Date: Thu, 27 Oct 2016 20:08:43 GMT
Server: BaseHTTP/0.3 Python/2.7.11

{
    "hello": "world"
}
```

The `chalice local` command *does not* assume the role associated with your lambda function, so you'll need to use an `AWS_PROFILE` that has sufficient permissions to your AWS resources used in your `app.py` .

Deleting Your App

You can use the `chalice delete` command to delete your app. Similar to the `chalice deploy` command, you can specify which chalice stage to delete. By default it will delete the `dev` stage:

```
$ chalice delete --stage dev
Deleting Rest API: duvw4kwyl3
Deleting function aws:arn:lambda:region:123456789:helloworld-dev
Deleting IAM Role helloworld-dev
```

Feedback

We'also love to hear from you. Please create any Github issues for additional features you'd like to see over at https://github.com/aws/chalice/issues. You can also chat with us on gitter: https://gitter.im/awslabs/chalice

FAQ

Q: How does the Python Serverless Microframework for AWS compare to other similar frameworks?

The biggest difference between this framework and others is that the Python Serverless Microframework for AWS is singularly focused on using a familiar, decorator-based API to write python applications that run on Amazon API Gateway and AWS Lambda. You can think of it as Flask/Bottle for serverless APIs. Its goal is to make writing and deploying these types of applications as simple as possible specifically for Python developers.

To achieve this goal, it has to make certain tradeoffs. Python will always remain the only supported language in this framework. Not every feature of API Gateway and Lambda is exposed in the framework. It makes assumptions about how applications will be deployed, and it has restrictions on how an application can be structured. It does not address the creation and lifecycle of other AWS resources your application may need (Amazon S3 buckets, Amazon DynamoDB tables, etc.). The feature set is purposefully small.

Other full-stack frameworks offer a lot more features and configurability than what this framework has and likely will ever have. Those frameworks are excellent choices for applications that need more than what is offered by this microframework. If all you need is to create a simple rest API in Python that runs on Amazon API Gateway and AWS Lambda, consider giving the Python Serverless Microframework for AWS a try.

Related Projects

- serverless - Build applications comprised of microservices that run in response to events, auto-scale for you, and only charge you when they run.
- Zappa - Deploy python WSGI applications on AWS Lambda and API Gateway.
- claudia - Deploy node.js projects to AWS Lambda and API Gateway.

- Domovoi - An extension to Chalice that handles a variety of AWS Lambda event sources such as SNS push notifications, S3 events, and Step Functions state machines.