Python Serverless Microframework for AWS

Release 1.2.2

Apr 13, 2018

Contents

The python serverless microframework for AWS allows you to quickly create and deploy applications that use Amazon API Gateway and AWS Lambda. It provides:

- A command line tool for creating, deploying, and managing your app
- A familiar and easy to use API for declaring views in python code
- Automatic IAM policy generation

```
$ pip install chalice
$ chalice new-project helloworld && cd helloworld
$ cat app.py

from chalice import Chalice

app = Chalice(app_name="helloworld")

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

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

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

Up and running in less than 30 seconds.

CHAPTER 1

Getting Started

Quickstart and Tutorial

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:

```
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:

```
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: 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:

```
@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.
- 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/2015-03-01'
                    '/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 lambda function helloworld-dev
Delete the role helloworld-dev? [y/N]: y
Deleting role name helloworld-dev
```

CHAPTER 2

Topics

Routing

The *Chalice.route()* method is used to contruct which routes you want to create for your API. The concept is the same mechanism used by Flask and bottle. You decorate a function with `@app.route(...)`, and whenever a user requests that URL, the function you've decorated is called. For example, suppose you deployed this app:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/')
def index():
    return {'view': 'index'}

@app.route('/a')
def a():
    return {'view': 'a'}

@app.route('/b')
def b():
    return {'view': 'b'}
```

If you go to `https://endpoint/`, the `index()` function would be called. If you went to `https://endpoint/a` and `https://endpoint/b`, then the `a()` and `b()` function would be called, respectively.

Note: Do not end your route paths with a trailing slash. If you do this, the `chalice deploy` command will raise a validation error.

You can also create a route that captures part of the URL. This captured value will then be passed in as arguments to your view function:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/users/{name}')
```

```
def users(name):
    return {'name': name}
```

If you then go to `https://endpoint/users/james`, then the view function will be called as: `users('james')`. The parameters are passed as keyword parameters based on the name as they appear in the URL. The argument names for the view function must match the name of the captured argument:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/a/{first}/b/{second}')
def users(first, second):
    return {'first': first, 'second': second}
```

Other Request Metadata

The route path can only contain `[a-zA-Z0-9._-]` chars and curly braces for parts of the URL you want to capture. You do not need to model other parts of the request you want to capture, including headers and query strings. Within a view function, you can introspect the current request using the *`app.current_request`* attribute. This also means you cannot control the routing based on query strings or headers. Here's an example for accessing query string data in a view function:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/users/{name}')
def users(name):
    result = {'name': name}
    if app.current_request.query_params.get('include-greeting') == 'true':
        result['greeting'] = 'Hello, %s' % name
    return result
```

In the function above, if the user provides a `?include-greeting=true` in the HTTP request, then an additional `greeting` key will be returned:

```
$ http https://endpoint/api/users/bob

{
    "name": "bob"
}

$ http https://endpoint/api/users/bob?include-greeting=true

{
    "greeting": "Hello, bob",
    "name": "bob"
}
```

Views

A view function in chalice is the function attached to an `@app.route()` decorator. In the example below, `index` is the view function:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


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

View Function Parameters

A view function's parameters correspond to the number of captured URL parameters specified in the `@app.route` call. In the example above, the route `/` specifies no captured parameters so the `index` view function accepts no parameters. However, in the view function below, a single URL parameter, `{city}` is specified, so the view function must accept a single parameter:

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/cities/{city}')
def index(city):
    return {'city': city}
```

This indicates that the value of `{city}` is variable, and whatever value is provided in the URL is passed to the `index` view function. For example:

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

If you want to access any other metdata of the incoming HTTP request, you can use the `app.current_request` property, which is an instance of the the *`Request`* class.

View Function Return Values

The response returned back to the client depends on the behavior of the view function. There are several options available:

- Returning an instance of *`Response`*. This gives you complete control over what gets returned back to the customer.
- A `bytes` type response body must have a `Content-Type` header value that is present in the `app.api.binary_types` list in order to be handled properly.
- Any other return value will be serialized as JSON and sent back as the response body with content type `application/json`.
- Any subclass of `ChaliceViewError` will result in an HTTP response being returned with the status code associated with that response, and a JSON response body containing a `Code` and a `Message`. This is discussed in more detail below.

- Any other exception raised will result in a 500 HTTP response. The body of that response depends on whether debug mode is enabled.

Error Handling

Chalice provides a built in set of exception classes that map to common HTTP errors including:

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

You can raise these anywhere in your view functions and chalice will convert these to the appropriate HTTP response. The default chalice error responses will send the error back as `application/json` with the response body containing a `Code` corresponding to the exception class name and a `Message` key corresponding to the string provided when the exception was instantiated. For example:

```
from chalice import Chalice
from chalice import BadRequestError

app = Chalice(app_name="badrequset")

@app.route('/badrequest')
def badrequest():
    raise BadRequestError("This is a bad request")
```

This view function will generate the following HTTP response:

```
$ http https://endpoint/api/badrequest
HTTP/1.1 400 Bad Request

{
    "Code": "BadRequestError",
    "Message": "This is a bad request"
}
```

In addition to the built in chalice exceptions, you can use the *`Response`* class to customize the HTTP errors if you prefer to either not have JSON error responses or customize the JSON response body for errors. For example:

```
from chalice import Chalice, Response

app = Chalice(app_name="badrequest")

@app.route('/badrequest')
def badrequest():
    return Response(body='Plain text error message',
                    headers={'Content-Type': 'text/plain'},
                    status_code=400)
```

Specifying HTTP Methods

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` as shown below:

```
POST /myview   --> myview()
PUT /myview  --> 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 as shown below:

```
POST /myview   --> myview_post()
PUT /myview  --> myview_put()
```

If you do chose to use separate view functions for the same route path, it is important to know:

- View functions that share the same route cannot have the same names. For example, two view functions that both share the same route path cannot both be named `view()`.
- View functions that share the same route cannot overlap in supported HTTP methods. For example if two view function both share the same route path, they both cannot contain `'PUT'` in their route `methods` list.
- View functions that share the same route path and have CORS configured cannot have differing CORS configuration. For example, if two view functions that both share the same route path, the route configuration for one of the view functions cannot set `cors=True` while having the route configuration of the other view function be set to `cors=app.CORSConfig(allow_origin='https://foo.example.com')`.

Binary Content

Chalice supports binary payloads through its `app.api.binary_types` list. Any type in this list is considered a binary `Content-Type`. Whenever a request with a `Content-Type` header is encountered that matches an entry in the `binary_types` list, its body will be available as a `bytes` type on the property `app.current_request.raw_body`. Similarly, in order to send binary data back in a response, simply set your `Content-Type` header to something present in the `binary_types` list. Note that you can override the default types by modifying the `app.api.binary_types` list at the module level.

Here is an example app which simply echos back binary content:

```
from chalice import Chalice, Response

app = Chalice(app_name="binary-response")

@app.route('/bin-echo', methods=['POST'],
           content_types=['application/octet-stream'])
def bin_echo():
    raw_request_body = app.current_request.raw_body
    return Response(body=raw_request_body,
                    status_code=200,
                    headers={'Content-Type': 'application/octet-stream'})
```

You can see this app echo back binary data sent to it:

```
$ echo -n -e "\xFE\xED" | http POST $(chalice url)bin-echo \
  Accept:application/octet-stream Content-Type:application/octet-stream | xxd
0000000: feed                                      ..
```

Note that both the `Accept` and `Content-Type` header are required. If you fail to set the `Content-Type` header on the request will result in a `415 UnsupportedMediaType` error. Care must be taken when configuring what `content_types` a route accepts, they must all be valid binary types, or they must all be non-binary types. The `Accept` header must also be set if the data returned is to be the raw binary, if is omitted the call return a `400` Bad Request response.

For example, here is the same call as above without the `Accept` header:

```
$ echo -n -e "\xFE\xED" | http POST  $(chalice url)bin-echo \
  Content-Type:application/octet-stream
HTTP/1.1 400 Bad Request
Connection: keep-alive
Content-Length: 270
Content-Type: application/json
Date: Sat, 27 May 2017 07:09:51 GMT

{
  "Code": "BadRequest",
  "Message": "Request did not specify an Accept header with
    application/octet-stream, The response has a Content-Type of
    application/octet-stream. If a response has a binary Content-Type then
    the request must specify an Accept header that matches."
}
```

Usage Recommendations

If you want to return a JSON response body, just return the corresponding python types directly. You don't need to use the *Response* class. Chalice will automatically convert this to a JSON HTTP response as a convenience for you.

Use the *Response* class when you want to return non-JSON content, or when you want to inject custom HTTP headers to your response.

For errors, raise the built in `ChaliceViewError` subclasses (e.g `BadRequestError`, `NotFoundError`, `ConflictError` etc) when you want to return a HTTP error response with a preconfigured JSON body containing a `Code` and `Message`.

Use the *Response* class when you want to customize the error responses to either return a different JSON error response body, or to return an HTTP response that's not `application/json`.

Configuration File

Whenever you create a new project using `chalice new-project`, a `.chalice` directory is created for you. In this directory is a `config.json` file that you can use to control what happens when you `chalice deploy`:

```
$ tree -a
.
-- .chalice
|   -- config.json
-- app.py
-- requirements.txt

1 directory, 3 files
```

Stage Specific Configuration

As of version 0.7.0 of chalice, you can specify configuration that is specific to a chalice stage as well as configuration that should be shared across all stages. See the *Chalice Stages* doc for more information about chalice stages.

- `stages` - This value of this key is a mapping of chalice stage name to stage configuration. Chalice assumes a default stage name of `dev`. If you run the `chalice new-project` command on chalice 0.7.0 or higher, this key along with the default `dev` key will automatically be created for you. See the examples section below for some stage specific configurations.

The following config values can either be specified per stage config or as a top level key which is not tied to a specific key. Whenever a stage specific configuration value is needed, the `stages` mapping is checked first. If no value is found then the top level keys will be checked.

- `api_gateway_stage` - The name of the API gateway stage. This will also be the URL prefix for your API (`https://endpoint/prefix/your-api`).
- `manage_iam_role` - `true`/`false`. Indicates if you want chalice to create and update the IAM role used for your application. By default, this value is `true`. However, if you have a pre-existing role you've created, you can set this value to `false` and a role will not be created or updated. `"manage_iam_role": false` means that you are responsible for managing the role and any associated policies associated with that role. If this value is `false` you must specify an `iam_role_arn`, otherwise an error is raised when you try to run `chalice deploy`.
- `iam_role_arn` - If `manage_iam_role` is `false`, you must specify this value that indicates which IAM role arn to use when configuration your application. This value is only used if `manage_iam_role` is `false`.
- `autogen_policy` - A boolean value that indicates if chalice should try to automatically generate an IAM policy based on analyzing your application source code. The default value is `true`. If this value is `false` then chalice will load try to a local file in `.chalice/policy-<stage-name>.json` instead of auto-generating a policy from source code analysis.

- `iam_policy_file` - When `autogen_policy` is false, chalice will try to load an IAM policy from disk instead of auto-generating one based on source code analysis. The default location of this file is `.chalice/policy-<stage-name>.json`, e.g `.chalice/policy-dev.json`, `.chalice/policy-prod.json`, etc. You can change the filename by providing this `iam_policy_file` config option. This filename is relative to the `.chalice` directory.
- `environment_variables` - A mapping of key value pairs. These key value pairs will be set as environment variables in your application. All environment variables must be strings. If this key is specified in both a stage specific config option as well as a top level key, the stage specific environment variables will be merged into the top level keys. See the examples section below for a concrete example.
- `lambda_timeout` - An integer representing the function execution time, in seconds, at which AWS Lambda should terminate the function. The default `lambda_timeout` is `60` seconds.
- `lambda_memory_size` - An integer representing the amount of memory, in MB, your Lambda function is given. AWS Lambda uses this memory size to infer the amount of CPU allocated to your function. The default `lambda_memory_size` value is `128`. The value must be a multiple of 64 MB.
- `tags` - A mapping of key value pairs. These key value pairs will be set as the tags on the resources running your deployed application. All tag keys and values must be strings. Similar to `environment_variables`, if a key is specified in both a stage specific config option as well as a top level key, the stage specific tags will be merged into the top level keys. By default, all chalice deployed resources are tagged with the key `'aws-chalice'` whose value is `'version={chalice-version}:stage={stage-name}:app={app-name}'`. Currently only the following chalice deployed resources are tagged: Lambda functions.

Examples

Here's an example for configuring IAM policies across stages:

```
{
  "version": "2.0",
  "app_name": "app",
  "stages": {
    "dev": {
      "autogen_policy": true,
      "api_gateway_stage": "dev"
    },
    "beta": {
      "autogen_policy": false,
      "iam_policy_file": "beta-app-policy.json"
    },
    "prod": {
      "manage_iam_role": false,
      "iam_role_arn": "arn:aws:iam::...:role/prod-role"
    }
  }
}
```

In this config file we're specifying three stages, `dev`, `beta`, and `prod`. In the `dev` stage, chalice will automatically generate an IAM policy based on analyzing the application source code. For the `beta` stage, chalice will load the `.chalice/beta-app-policy.json` file and use it as the policy to associate with the IAM role for that stage. In the `prod` stage, chalice won't modify any IAM roles. It will just set the IAM role for the Lambda function to be `arn:aws:iam::...:role/prod-role`.

Here's an example that show config precedence:

```
{
  "version": "2.0",
  "app_name": "app",
  "api_gateway_stage": "api"
  "stages": {
    "dev": {
    },
    "beta": {
    },
    "prod": {
      "api_gateway_stage": "prod",
      "manage_iam_role": false,
      "iam_role_arn": "arn:aws:iam::...:role/prod-role"
    }
  }
}
```

In this config file, both the `dev` and `beta` stage will have an API gateway stage name of `api` because they will default to the top level `api_gateway_stage` key. However, the `prod` stage will have an API gateway stage name of `prod` because the `api_gateway_stage` is specified in `{"stages": {"prod": ...}}` mapping.

In the following example, environment variables are specified both as top level keys as well as per stage. This allows us to provide environment variables that all stages should have as well as stage specific environment variables:

```
{
  "version": "2.0",
  "app_name": "app",
  "environment_variables": {
    "SHARED_CONFIG": "foo"
    "OTHER_CONFIG": "from-top"
  }
  "stages": {
    "dev": {
      "environment_variables": {
        "TABLE_NAME": "dev-table",
        "OTHER_CONFIG": "dev-value"
      }
    },
    "prod": {
      "environment_variables": {
        "TABLE_NAME": "prod-table",
        "OTHER_CONFIG": "prod-value"
      }
    }
  }
}
```

For the above config, the `dev` stage will have the following environment variables set:

```
{
  "SHARED_CONFIG": "foo",
  "TABLE_NAME": "dev-table",
  "OTHER_CONFIG": "dev-value",
}
```

The `prod` stage will have these environment variables set:

```
{
  "SHARED_CONFIG": "foo",
```

```
  "TABLE_NAME": "prod-table",
  "OTHER_CONFIG": "prod-value",
}
```

It is also possible to add specific configurations for specific Lambda functions in a stage. Here is an example of a configuration for a function called `foo` in a `dev` stage:

```
{
  "stages": {
    "dev": {
      "api_gateway_stage": "api",
      "lambda_functions": {
        "foo": {
          "lambda_timeout": 120
        }
      }
    }
  },
  "version": "2.0",
  "app_name": "demo"
}
```

Multifile Support

The `app.py` file contains all of your view functions and route information, but you don't have to keep all of your application code in your `app.py` file.

As your application grows, you may reach out a point where you'd prefer to structure your application in multiple files. You can create a `chalicelib/` directory, and anything in that directory is recursively included in the deployment package. This means that you can have files besides just `.py` files in `chalicelib/`, including `.json` files for config, or any kind of binary assets.

Let's take a look at a few examples.

Consider the following app directory structure layout:

```
.
-- app.py
-- chalicelib
|   -- __init__.py
-- requirements.txt
```

Where `chalicelib/__init__.py` contains:

```
MESSAGE = 'world'
```

and the `app.py` file contains:

```
from chalice import Chalice
from chalicelib import MESSAGE

app = Chalice(app_name="multifile")

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

Note in line 2 we're importing the `MESSAGE` variable from the `chalicelib` package, which is a top level directory in our project. We've created a `chalicelib/__init__.py` file which turns the `chalicelib` directory into a python package.

We can also use this directory to store config data. Consider this app structure layout:

```
.
-- app.py
-- chalicelib
|   -- config.json
-- requirements.txt
```

With `chalicelib/config.json` containing:

```
{"message": "world"}
```

In our `app.py` code, we can load and use our config file:

```
1  import os
2  import json
3
4  from chalice import Chalice
5
6  app = Chalice(app_name="multifile")
7
8  filename = os.path.join(
9      os.path.dirname(__file__), 'chalicelib', 'config.json')
10 with open(filename) as f:
11     config = json.load(f)
12
13 @app.route("/")
14 def index():
15     # We can access ``config`` here if we want.
16     return {"hello": config['message']}
```

Logging

You have several options for logging in your application. You can use any of the options available to lambda functions as outlined in the AWS Lambda Docs. The simplest option is to just use print statements. Anything you print will be accessible in cloudwatch logs as well as in the output of the `chalice logs` command.

In addition to using the stdlib `logging` module directly, the framework offers a preconfigured logger designed to work nicely with Lambda. This is offered purely as a convenience, you can use `print` or the `logging` module directly if you prefer.

You can access this logger via the `app.log` attribute, which is a a logger specifically for your application. This attribute is an instance of `logging.getLogger(your_app_name_)` that's been preconfigured with reasonable defaults:

- StreamHandler associated with `sys.stdout`.
- Log level set to `logging.ERROR` by default. You can also manually set the logging level by setting `app.log.setLevel(logging.DEBUG)`.
- A logging formatter that displays the app name, level name, and message.

Examples

In the following application, we're using the application logger to emit two log messages, one at `DEBUG` and one at the `ERROR` level:

```
from chalice import Chalice

app = Chalice(app_name='demolog')


@app.route('/')
def index():
    app.log.debug("This is a debug statement")
    app.log.error("This is an error statement")
    return {'hello': 'world'}
```

If we make a request to this endpoint, and then look at `chalice logs` we'll see the following log message:

```
2016-11-06 20:24:25.490000 9d2a92 demolog - ERROR - This is an error statement
```

As you can see, only the `ERROR` level log is emitted because the default log level is `ERROR`. Also note the log message formatting. This is the default format that's been automatically configured. We can make a change to set our log level to debug:

```
from chalice import Chalice

app = Chalice(app_name='demolog')
# Enable DEBUG logs.
app.log.setLevel(logging.DEBUG)


@app.route('/')
def index():
    app.log.debug("This is a debug statement")
    app.log.error("This is an error statement")
    return {'hello': 'world'}
```

Now if we make a request to the `/` URL and look at the output of `chalice logs`, we'll see the following log message:

```
2016-11-07 12:29:15.714 431786 demolog - DEBUG - This is a debug statement
2016-11-07 12:29:15.714 431786 demolog - ERROR - This is an error statement
```

As you can see here, both the debug and error log message are shown.

SDK Generation

The `@app.route(...)` information you provide chalice allows it to create corresponding routes in API Gateway. One of the benefits of this approach is that we can leverage API Gateway's SDK generation process. Chalice offers a `chalice generate-sdk` command that will automatically generate an SDK based on your declared routes.

Note: The only supported language at this time is javascript.

Keep in mind that chalice itself does not have any logic for generating SDKs. The SDK generation happens service side in API Gateway, the `chalice generate-sdk` is just a high level wrapper around that functionality.

To generate an SDK for a chalice app, run this command from the project directory:

```
$ chalice generate-sdk /tmp/sdk
```

You should now have a generated javascript sdk in `/tmp/sdk`. API Gateway includes a `README.md` as part of its SDK generation which contains details on how to use the javascript SDK.

Example

Suppose we have the following chalice app:

```
from chalice import Chalice

app = Chalice(app_name='sdktest')

@app.route('/', cors=True)
def index():
    return {'hello': 'world'}

@app.route('/foo', cors=True)
def foo():
    return {'foo': True}

@app.route('/hello/{name}', cors=True)
def hello_name(name):
    return {'hello': name}

@app.route('/users/{user_id}', methods=['PUT'], cors=True)
def update_user(user_id):
    return {"msg": "fake updated user", "userId": user_id}
```

Let's generate a javascript SDK and test it out in the browser. Run the following command from the project dir:

```
$ chalice generate-sdk /tmp/sdkdemo
$ cd /tmp/sdkdemo
$ ls -la
-rw-r--r--   1 jamessar  r  3227 Nov 21 17:06 README.md
-rw-r--r--   1 jamessar  r  9243 Nov 21 17:06 apigClient.js
drwxr-xr-x   6 jamessar  r   204 Nov 21 17:06 lib
```

You should now be able to follow the instructions from API Gateway in the `README.md` file. Below is a snippet that shows how the generated javascript SDK methods correspond to the `@app.route()` calls in chalice.

```
<script type="text/javascript">
  // Below are examples of how the javascript SDK methods
  // correspond to chalice @app.routes()
  var apigClient = apigClientFactory.newClient();

  // @app.route('/')
  apigClient.rootGet().then(result => {
      document.getElementById('root-get').innerHTML = JSON.stringify(result.data);
  });

  // @app.route('/foo')
  apigClient.fooGet().then(result => {
      document.getElementById('foo-get').innerHTML = JSON.stringify(result.data);
  });
```

```
  // @app.route('/hello/{name}')
  apigClient.helloNameGet({name: 'jimmy'}).then(result => {
      document.getElementById('helloname-get').innerHTML = JSON.stringify(result.data);
  });

  // @app.route('/users/{user_id}', methods=['PUT'])
  apigClient.usersUserIdPut({user_id: '123'}, 'body content').then(result => {
      document.getElementById('users-userid-put').innerHTML = JSON.stringify(result.data);
  });
</script>
```

Example HTML File

If you want to try out the example above, you can use the following index.html page to test:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>SDK Test</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/skeleton/2.0.4/skeleton.m
        <script type="text/javascript" src="lib/axios/dist/axios.standalone.js"></script>
        <script type="text/javascript" src="lib/CryptoJS/rollups/hmac-sha256.js"></script>
        <script type="text/javascript" src="lib/CryptoJS/rollups/sha256.js"></script>
        <script type="text/javascript" src="lib/CryptoJS/components/hmac.js"></script>
        <script type="text/javascript" src="lib/CryptoJS/components/enc-base64.js"></script>
        <script type="text/javascript" src="lib/url-template/url-template.js"></script>
        <script type="text/javascript" src="lib/apiGatewayCore/sigV4Client.js"></script>
        <script type="text/javascript" src="lib/apiGatewayCore/apiGatewayClient.js"></script>
        <script type="text/javascript" src="lib/apiGatewayCore/simpleHttpClient.js"></script>
        <script type="text/javascript" src="lib/apiGatewayCore/utils.js"></script>
        <script type="text/javascript" src="apigClient.js"></script>


        <script type="text/javascript">
          // Below are examples of how the javascript SDK methods
          // correspond to chalice @app.routes()
          var apigClient = apigClientFactory.newClient();

          // @app.route('/')
          apigClient.rootGet().then(result => {
              document.getElementById('root-get').innerHTML = JSON.stringify(result.data);
          });

          // @app.route('/foo')
          apigClient.fooGet().then(result => {
              document.getElementById('foo-get').innerHTML = JSON.stringify(result.data);
          });

          // @app.route('/hello/{name}')
          apigClient.helloNameGet({name: 'jimmy'}).then(result => {
              document.getElementById('helloname-get').innerHTML = JSON.stringify(result.data);
          });

          // @app.route('/users/{user_id}', methods=['PUT'])
          apigClient.usersUserIdPut({user_id: '123'}, 'body content').then(result => {
```

```
                document.getElementById('users-userid-put').innerHTML = JSON.stringify(result.data);
            });
        </script>
    </head>
    <body>
        <div><h5>result of rootGet()</h5><pre id="root-get"></pre></div>
        <div><h5>result of fooGet()</h5><pre id="foo-get"></pre></div>
        <div><h5>result of helloNameGet({name: 'jimmy'})</h5><pre id="helloname-get"></pre></div>
        <div><h5>result of usersUserIdPut({user_id: '123'})</h5><pre id="users-userid-put"></pre></di
    </body>
</html>
```

Chalice Stages

Chalice has the concept of stages, which are completely separate sets of AWS resources. When you first create a chalice project and run commands such as `chalice deploy` and `chalice url`, you don't have to specify any stage values or stage configuration. This is because chalice will use a stage named `dev` by default.

You may eventually want to have multiple stages of your application. A common configuration would be to have a `dev`, `beta` and `prod` stage. A `dev` stage would be used by developers to test out new features. Completed features would be deployed to `beta`, and the `prod` stage would be used for serving production traffic.

Chalice can help you manage this.

To create a new chalice stage, specify the `--stage` argument. If the stage does not exist yet, it will be created for you:

```
$ chalice deploy --stage prod
```

By creating a new chalice stage, a new API Gateway rest API, Lambda function, and potentially (depending on config settings) a new IAM role will be created for you.

Example

Let's say we have a new app:

```
$ chalice new-project myapp
$ cd myapp
$ chalice deploy
...
https://mmnkdi.execute-api.us-west-2.amazonaws.com/api/
```

We've just created our first stage, `dev`. We can iterate on our application and continue to run `chalice deploy` to deploy our code to the `dev` stage. Let's say we want to now create a `prod` stage. To do this, we can run:

```
$ chalice deploy --stage prod
...
https://wk9fhx.execute-api.us-west-2.amazonaws.com/api/
```

We now have two completely separate rest APIs:

```
$ chalice url --stage dev
https://mmnkdi.execute-api.us-west-2.amazonaws.com/api/

$ chalice url --stage prod
https://wk9fhx.execute-api.us-west-2.amazonaws.com/api/
```

Additionally, we can see all our deployed values by looking at the `.chalice/deployed.json` file:

```
$ cat .chalice/deployed.json
{
  "dev": {
    "region": "us-west-2",
    "api_handler_name": "myapp-dev",
    "api_handler_arn": "arn:aws:lambda:...:function:myapp",
    "rest_api_id": "wk9fhx",
    "chalice_version": "0.7.0",
    "api_gateway_stage": "dev",
    "backend": "api"
  },
  "prod": {
    "rest_api_id": "mmnkdi",
    "chalice_version": "0.7.0",
    "region": "us-west-2",
    "backend": "api",
    "api_handler_name": "myapp-prod",
    "api_handler_arn": "arn:aws:lambda:...:function:myapp-prod",
    "api_gateway_stage": "dev"
  }
}
```

App Packaging

In order to deploy your Chalice app, a zip file is created that contains your application and all third party packages your application requires. This file is used by AWS Lambda and is referred to as a deployment package.

Chalice will automatically create this deployment package for you, and offers several features to make this easier to manage. Chalice allows you to clearly separate application specific modules and packages you are writing from 3rd party package dependencies.

App Directories

You have two options to structure application specific code/config:

- **app.py** - This file includes all your route information and is always included in the deployment package.
- **chalicelib/** - This directory (if it exists) is included in the deployment package. This is where you can add config files and additional application modules if you prefer not to have all your app code in the `app.py` file.

See *Multifile Support* for more info on the `chalicelib/` directory. Both the `app.py` and the `chalicelib/` directory are intended for code that you write yourself.

3rd Party Packages

There are two options for handling python package dependencies:

- **requirements.txt** - During the packaging process, Chalice will install any packages it finds or can build compatible wheels for. Specifically all pure python packages as well as all packages that upload wheel files for the `manylinux1_x86_64` platform will be automatically installable.
- **vendor/** - The *contents* of this directory are automatically added to the top level of the deployment package.

Chalice will also check for an optional `vendor/` directory in the project root directory. The contents of this directory are automatically included in the top level of the deployment package (see *Examples* for specific examples). The `vendor/` directory is helpful in these scenarios:

- You need to include custom packages or binary content that is not accessible via `pip`. These may be internal packages that aren't public.
- Wheel files are not available for a package you need from pip.
- A package is installable with `requirements.txt` but has optional c extensions. Chalice can build the dependency without the c extensions, but if you want better performance you can vendor a version that is compiled.

As a general rule of thumb, code that you write goes in either `app.py` or `chalicelib/`, and dependencies are either specified in `requirements.txt` or placed in the `vendor/` directory.

Examples

Suppose I have the following app structure:

```
.
-- app.py
-- chalicelib
|   -- __init__.py
|   -- utils.py
-- requirements.txt
-- vendor
    -- internalpackage
        -- __init__.py
```

And the `requirements.txt` file had one requirement:

```
$ cat requirements.txt
sortedcontainers==1.5.4
```

Then the final deployment package directory structure would look like this:

```
.
-- app.py
-- chalicelib
|   -- __init__.py
|   -- utils.py
-- internalpackage
|   -- __init__.py
-- sortedcontainers
    -- __init__.py
```

This directory structure is then zipped up and sent to AWS Lambda during the deployment process.

Cryptography Example

> Note
>
> Since the original version of this example was written, cryptography has released version 2.0 which does have manylinux1 wheels available. This means if you want to use cryptography in a Chalice app all you need to do is add `cryptography` or `cryptography>=2.0` in your requirements.txt file.
>
> This example will use version 1.9 of Cryptography because it is a good example of a library with C extensions and no wheel files.

Below shows an example of how to use the cryptography 1.9 package in a Chalice app for the `python3.6` lambda environment.

Suppose you are on a Mac or Windows and want to deploy a Chalice app that depends on the `cryptography==1.9` package. If you simply add it to your `requirements.txt` file and try to deploy it with `chalice deploy` you will get the following warning during deployment:

```
$ cat requirements.txt
cryptography==1.9
$ chalice deploy
Updating IAM policy.
Updating lambda function...
Creating deployment package.

Could not install dependencies:
cryptography==1.9
You will have to build these yourself and vendor them in
the chalice vendor folder.

Your deployment will continue but may not work correctly
if missing dependencies are not present. For more information:
http://chalice.readthedocs.io/en/latest/topics/packaging.html
```

This happened because the `cryptography` version 1.9 does not have wheel files availble on PyPi, and has C extensions. Since we are not on the same platform as AWS Lambda, the compiled C extensions Chalice built were not compatible. To get around this we are going to leverage the `vendor/` directory, and build the `cryptography` package on a compatible linux system.

You can do this yourself by building `cryptography` on an Amazon Linux instance running in EC2. All of the following commands were run inside a `python 3.6` virtual environment.

- Download the source first:

```
$ pip download cryptography==1.9
```

  This will download all the requirements into the current working directory. The directory should have the following contents:

  - `asn1crypto-0.22.0-py2.py3-none-any.whl`
  - `cffi-1.10.0-cp36-cp36m-manylinux1_x86_64.whl`
  - `cryptography-1.9.tar.gz`
  - `idna-2.5-py2.py3-none-any.whl`
  - `pycparser-2.17.tar.gz`
  - `six-1.10.0-py2.py3-none-any.whl`

  This is a complete set of dependencies required for the cryptography package. Most of these packages have wheels that were downloaded, which means they can simply be put in the `requirements.txt` and Chalice will take care of downloading them. That leaves `cryptography` itself and `pycparser` as the only two that did not have a wheel file available for download.

- Next build the `cryptography` source package into a wheel file:

```
$ pip wheel cryptography-1.9.tar.gz
```

  This will take a few seconds and build a wheel file for both `cryptography` and `pycparser`. The directory should now have two additional wheel files:

  - `cryptography-1.9-cp36-cp36m-linux_x86_64.whl`

  - `pycparser-2.17-py2.py3-none-any.whl`

  The `cryptography` wheel file has been built with a compatible architecture for Lambda (`linux_x86_64`) and the `pycparser` has been built for `any` architecture which means it can also be automatically packaged by Chalice if it is listed in the `requirements.txt` file.

- Download the `cryptography` wheel file from the Amazon Linux instance and unzip it into the `vendor/` directory in the root directory of your Chalice app.

  You should now have a project directory that looks like this:

```
$ tree
.
-- app.py
-- requirements.txt
-- vendor
    -- cryptography
    |   -- ... Lots of files
    |
    -- cryptography-1.9.dist-info
        -- DESCRIPTION.rst
        -- METADATA
        -- RECORD
        -- WHEEL
        -- entry_points.txt
        -- metadata.json
        -- top_level.txt
```

  The `requirements.txt` file should look like this:

```
$ cat requirements.txt
cffi==1.10.0
six==1.10.0
asn1crypto==0.22.0
idna==2.5
pycparser==2.17
```

  In your `app.py` file you can now import `cryptography`, and these dependencies will all get included when the `chalice deploy` command is run.

Python Version Support

Chalice supports all versions of python supported by AWS Lambda, which is currently `python2.7` and `python3.6`.

Chalice will automatically pick which version of python to use for Lambda based on the major version of python you are using. You don't have to explicitly configure which version of python you want to use. For example:

```
$ python --version
Python 3.6.1
$ chalice new-project test-versions
$ cd test-versions
$ chalice package test-package
$ grep -C 3 python test-package/sam.json
    "APIHandler": {
      "Type": "AWS::Serverless::Function",
      "Properties": {
        "Runtime": "python3.6",
```

```
        "Handler": "app.app",
        "CodeUri": "./deployment.zip",
        "Events": {

# Similarly, if we were to run "chalice deploy" we'd
# use python3.6 for the runtime.
$ chalice --debug deploy
Initiating first time deployment...
Deploying to: dev
...
"Runtime":"python3.6"
...
https://rest-api-id.execute-api.us-west-2.amazonaws.com/api/
```

In the example above, we're using python 3.6.1 so chalice automatically selects the `python3.6` runtime for lambda. If we were using python 2.7.11, chalice would automatically select `python2.7` as the runtime.

Chalice will emit a warning if the minor version does not match a python version supported by Lambda. Chalice will select the closest Lambda version in this scenario, as shown in the table below.

Local Python Version	Lambda Python Runtime
python2.7.10	python2.7
python2.7.11	python2.7
python2.7.12	python2.7
python2.7.13	python2.7
python3.3.6	python3.6
python3.4.6	python3.6
python3.5.3	python3.6
python3.6.0	python3.6
python3.6.1	python3.6

We strongly encourage you to develop your application using the same major/minor version of python you plan on using on AWS Lambda.

Changing Python Runtime Versions

The version of the python runtime to use in AWS Lambda can be reconfigured whenever you deploy your chalice app. This allows you to migrate to python3 in AWS Lambda by creating a new virtual environment that uses python3. For example, suppose you have an existing chalice app that uses python2:

```
$ python --version
Python 2.7.12
$ chalice deploy
...
https://endpoint/api
```

To upgrade the application to use python3, create a python3 virtual environment and redeploy. When this happens, you will be prompted to confirm the python runtime version changing:

```
$ deactivate
$ virtualenv --python python3 /tmp/venv3
$ source /tmp/venv3/bin/activate
$ python --version
Python 3.6.1
$ chalice deploy
...
```

```
The python runtime will change from python2.7 to python3.6,
would you like to continue?  [Y/n]: y
...
https://endpoint/api
```

AWS CloudFormation Support

When you run `chalice deploy`, chalice will deploy your application using the AWS SDK for Python). Chalice also provides functionality that allows you to manage deployments yourself using cloudformation. This is provided via the `chalice package` command.

When you run this command, chalice will generate the AWS Lambda deployment package that contains your application as well as a Serverless Application Model (SAM) template. You can then use a tool like the AWS CLI, or any cloudformation deployment tools you use, to deploy your chalice application.

Considerations

Using the `chalice package` command is useful when you don't want to use `chalice deploy` to manage your deployments. There's several reasons why you might want to do this:

- You have pre-existing infrastructure and tooling set up to manage cloudformation stacks.
- You want to integrate with other cloudformation stacks to manage all your AWS resources, including resources outside of your chalice app.
- You'd like to integrate with AWS CodePipeline to automatically deploy changes when you push to a git repo.

Keep in mind that you can't switch between `chalice deploy` and `chalice package` + CloudFormation for deploying your app.

If you choose to use `chalice package` and CloudFormation to deploy your app, you won't be able to switch back to `chalice deploy`. Running `chalice deploy` would create an entirely new set of AWS resources (API Gateway Rest API, AWS Lambda function, etc).

Example

In this example, we'll create a chalice app and deploy it using the AWS CLI.

First install the necessary packages:

```
$ virtualenv /tmp/venv
$ . /tmp/venv/bin/activate
$ pip install chalice awscli
$ chalice new-project test-cfn-deploy
$ cd test-cfn-deploy
```

At this point we've installed chalice and the AWS CLI and we have a basic app created locally. Next we'll run the `package` command and look at its contents:

```
$ $ chalice package /tmp/packaged-app/
Creating deployment package.
$ ls -la /tmp/packaged-app/
-rw-r--r--   1 j         wheel  3355270 May 25 14:20 deployment.zip
-rw-r--r--   1 j         wheel     3068 May 25 14:20 sam.json
```

```
$ unzip -l /tmp/packaged-app/deployment.zip  | tail -n 5
    17292  05-25-17 14:19   chalice/app.py
      283  05-25-17 14:19   chalice/__init__.py
      796  05-25-17 14:20   app.py
 --------                   -------
  9826899                   723 files

$ head < /tmp/packaged-app/sam.json
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Outputs": {
    "RestAPIId": {
      "Value": {
        "Ref": "RestAPI"
      }
    },
    "APIHandlerName": {
      "Value": {
```

As you can see in the above example, the `package` command created a directory that contained two files, a `deployment.zip` file, which is the Lambda deployment package, and a `sam.json` file, which is the SAM template that can be deployed using CloudFormation. Next we're going to use the AWS CLI to deploy our app. To this, we'll first run the `aws cloudformation package` command, which will take our deployment.zip file and upload to an S3 bucket we specify:

```
$ aws cloudformation package \
     --template-file /tmp/packaged-app/sam.json \
     --s3-bucket myapp-bucket \
     --output-template-file /tmp/packaged-app/packaged.yaml
```

Now we can deploy our app using the `aws cloudformation deploy` command:

```
$ aws cloudformation deploy \
    --template-file /tmp/packaged-app/packaged.yaml \
    --stack-name test-cfn-stack \
    --capabilities CAPABILITY_IAM
Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - test-cfn-stack
```

This will take a few minutes to complete, but once it's done, the endpoint url will be available as an output:

```
$ aws cloudformation describe-stacks --stack-name test-cfn-stack \
  --query "Stacks[].Outputs[?OutputKey=='EndpointURL'][] | [0].OutputValue"
"https://abc29hkq0i.execute-api.us-west-2.amazonaws.com/api/"

$ http "https://abc29hkq0i.execute-api.us-west-2.amazonaws.com/api/"
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 18
Content-Type: application/json
...

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

Authorization

Chalice supports multiple mechanisms for authorization. This topic covers how you can integrate authorization into your Chalice applications.

In Chalice, all the authorizers are configured per-route and specified using the `authorizer` kwarg to an `@app.route()` call. You control which type of authorizer to use based on what's passed as the `authorizer` kwarg. You can use the same authorizer instance for multiple routes.

The first set of authorizers chalice supports cover the scenario where you have some existing authorization mechanism that you just want your Chalice app to use.

Chalice also supports built-in authorizers, which allows Chalice to manage your custom authorizers as part of `chalice deploy`. This is covered in the Built-in Authorizers section.

AWS IAM Authorizer

The IAM Authorizer allows you to control access to API Gateway with IAM permissions

To associate an IAM authorizer with a route in chalice, you use the `IAMAUthorizer` class:

```
from chalice import IAMAuthorizer

authorizer = IAMAuthorizer()

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

See the API Gateway documentation for more information on controlling access to API Gateway with IAM permissions.

Amazon Cognito User Pools

In addition to using IAM roles and policies with the *`IAMAuthorizer`* you can also use a Cognito user pools to control who can access your Chalice app. A cognito user pool serves as your own identity provider to maintain a user directory.

To integrate Cognito user pools with Chalice, you'll need to have an existing cognito user pool configured.

```
from chalice import CognitoUserPoolAuthorizer

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

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

For more information about using Cognito user pools with API Gateway, see the Use Amazon Cognito User Pools documentation.

Custom Authorizers

API Gateway also lets you write custom authorizers using a Lambda function. You can configure a Chalice route to use a pre-existing Lambda function as a custom authorizer. If you also want to write and manage your Lambda

authorizer using Chalice, see the next section, Built-in Authorizers.

To connect an existing Lambda function as a custom authorizer in chalice, you use the `CustomAuthorizer` class:

```
from chalice import CustomAuthorizer

authorizer = CustomAuthorizer(
    'MyCustomAuth', header='Authorization',
    authorizer_uri=('arn:aws:apigateway:region:lambda:path/2015-03-01'
                    '/functions/arn:aws:lambda:region:account-id:'
                    'function:FunctionName/invocations'))

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

Built-in Authorizers

The `IAMAuthorizer`, `CognitoUserPoolAuthorizer`, and the `CustomAuthorizer` classes are all for cases where you have existing resources for managing authorization and you want to wire them together with your Chalice app. A Built-in authorizer is used when you'd like to write your custom authorizer in Chalice, and have the additional Lambda functions managed when you run `chalice deploy/delete`. This section will cover how to use the built-in authorizers in chalice.

Creating an authorizer in chalice requires you use the `@app.authorizer` decorator to a function. The function must accept a single arg, which will be an instance of *`AuthRequest`*. The function must return a *`AuthResponse`*. As an example, we'll port the example from the API Gateway documentation. First, we'll show the code and then walk through it:

```
from chalice import Chalice, AuthResponse

app = Chalice(app_name='demoauth1')


@app.authorizer()
def demo_auth(auth_request):
    token = auth_request.token
    # This is just for demo purposes as shown in the API Gateway docs.
    # Normally you'd call an oauth provider, validate the
    # jwt token, etc.
    # In this exampe, the token is treated as the status for demo
    # purposes.
    if token == 'allow':
        return AuthResponse(routes=['/'], principal_id='user')
    else:
        # By specifying an empty list of routes,
        # we're saying this user is not authorized
        # for any URLs, which will result in an
        # Unauthorized response.
        return AuthResponse(routes=[], principal_id='user')


@app.route('/', authorizer=demo_auth)
def index():
    return {'context': app.current_request.context}
```

In the example above we define a built-in authorizer by decorating the `demo_auth` function with the `@app.authorizer()` decorator. Note you must use `@app.authorizer()` and not `@app.authorizer`. A

built-in authorizer function has this type signature:

```
def auth_handler(auth_request: AuthRequest) -> AuthResponse: ...
```

Within the auth handler you must determine if the request is authorized or not. The `AuthResponse` contains the allowed URLs as well as the principal id of the user. You can optionally return a dictionary of key value pairs (as the `context` kwarg). This dictionary will be passed through on subsequent requests. In our example above we're not using the context dictionary.

Now let's deploy our app. As usual, we just need to run `chalice deploy` and chalice will automatically deploy all the necessary Lambda functions for us.

Now when we try to make a request, we'll get an Unauthorized error:

```
$ http https://api.us-west-2.amazonaws.com/api/
HTTP/1.1 401 Unauthorized

{
    "message": "Unauthorized"
}
```

If we add the appropriate authorization header, we'll see the call succeed:

```
$ http https://api.us-west-2.amazonaws.com/api/ 'Authorization: allow'
HTTP/1.1 200 OK

{
    "context": {
        "accountId": "12345",
        "apiId": "api",
        "authorizer": {
            "principalId": "user"
        },
        "httpMethod": "GET",
        "identity": {
            "accessKey": null,
            "accountId": null,
            "apiKey": "",
            "caller": null,
            "cognitoAuthenticationProvider": null,
            "cognitoAuthenticationType": null,
            "cognitoIdentityId": null,
            "cognitoIdentityPoolId": null,
            "sourceIp": "1.1.1.1",
            "user": null,
            "userAgent": "HTTPie/0.9.9",
            "userArn": null
        },
        "path": "/api/",
        "requestId": "d35d2063-56be-11e7-9ce1-dd61c24a3668",
        "resourceId": "id",
        "resourcePath": "/",
        "stage": "dev"
    }
}
```

The low level API for API Gateway's custom authorizer feature requires that an IAM policy must be returned. The *AuthResponse* class we're using is a wrapper over building the IAM policy ourself. If you want low level control and would prefer to contruct the IAM policy yourself you can return a dictionary of the IAM policy instead of an instance of *AuthResponse*. If you do that, the dictionary is returned without modification back to API Gateway.

For more information on custom authorizers, see the Use API Gateway Custom Authorizers page in the API Gateway user guide.

Lambda Event Sources

Scheduled Events

Chalice has support for *scheduled events*. This feature allows you to periodically invoke a lambda function based on some regular schedule. You can specify a fixed rate or a cron expression.

To create a scheduled event in chalice, you use the `@app.schedule()` decorator. Let's look at an example.

```
app = chalice.Chalice(app_name='foo')

@app.schedule('rate(1 hour)')
def every_hour(event):
    print(event.to_dict())

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

In this example, we've updated the starter hello world app with a scheduled event. When you run `chalice deploy` Chalice will create two Lambda functions. The first lambda function is for the API handler used by API gateway. The second lambda function will be for the scheduled CloudWatch event (the `every_hour` function). The `every_hour` function will be automatically invoked every hour by Lambda.

The *`Chalice.schedule()`* method accepts either a string or an instance of *`Rate`* or *`Cron`*. For example:

```
app = chalice.Chalice(app_name='foo')

@app.schedule(Rate(1, unit=Rate.HOURS))
def every_hour(event):
    print(event.to_dict())
```

The function you decorate must accept a single argument, which will be of type *`CloudWatchEvent`*.

Limitations:

- You must provide at least 1 `@app.route` decorator. It is not possible to deploy only scheduled events without an API Gateway API.

Pure Lambda Functions

Chalice provides abstractions over AWS Lambda functions, including:

- An API handler the coordinates with API Gateway for creating rest APIs.
- A custom authorizer that allows you to integrate custom auth logic in your rest API.
- A scheduled event that includes managing the CloudWatch Event rules, targets, and permissions.

However, chalice also supports managing pure Lambda functions that don't have any abstractions built on top. This is useful if you want to create a Lambda function for something that's not supported by chalice or if you just want to create Lambda functions but don't want to manage handling dependencies and deployments yourself.

In order to do this, you can use the *Chalice.lambda_function()* decorator to denote that this python function is a pure lambda function that should be invoked as is, without any input or output mapping. When you use this function, you must provide a function that maps to the same function signature expected by AWS Lambda as defined here.

Let's look at an example.

```
app = chalice.Chalice(app_name='foo')

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

@app.lambda_function()
def custom_lambda_function(event, context):
    # Anything you want here.
    return {}

@app.lambda_function(name='MyFunction')
def other_lambda_function(event, context):
    # Anything you want here.
    return {}
```

In this example, we've updated the starter hello world app with two extra Lambda functions. When you run `chalice deploy` Chalice will create three Lambda functions. The first lambda function is for the API handler used by API gateway. The second and third lambda function will be pure lambda functions. These two additional lambda functions won't be hooked up to anything. You'll need to manage connecting them to any additional AWS Resources on your own.

Continuous Deployment (CD)

Chalice can be used to set up a basic Continuous Deployment pipeline. The `chalice deploy` command is good for getting up and running quickly with Chalice, but in a team environment properly managing permissions and sharing and updating the `deployed.json` file will get messy.

One way to scale up your chalice app is to create a continuous deployment pipeline. The pipeline can run tests on code changes and, if they pass, promote the new build to a testing stage. More checks can be put in place to manually promote a build to production, or you can do so automatically. This model greatly simplifies managing what resources belong to your Chalice app as they are all stored in the Continuous Deployment pipeline.

Chalice can generate a CloudFormation template that will create a starter CD pipeline. It contains a CodeCommit repo, a CodeBuild stage for packaging your chalice app, and a CodePipeline stage to deploy your application using CloudFormation.

Usage example

Setting up the deployment pipeline is a two step process. First use the `chalice generate-pipeline` command to generate a base CloudFormation template. Second use the AWS CLI to deploy the CloudFormation template using the `aws cloudformation deploy` command. Below is an example.

```
$ chalice generate-pipeline pipeline.json
$ aws cloudformation deploy --stack-name mystack
      --template-file pipeline.json --capabilities CAPABILITY_IAM
Waiting for changeset to be created..
```

```
Waiting for stack create/update to complete
Successfully created/updated stack - mystack
```

Once the CloudFormation template has finished creating the stack, you will have several new AWS resources that make up a bare bones CD pipeline.

- **CodeCommit Repository** - The CodeCommit repository is the entrypoint into the pipeline. Any code you want to deploy should be pushed to this remote.
- **CodePipeline Pipeline** - The CodePipeline is what coordinates the build process, and pushes the released code out.
- **CodeBuild Project** - The CodeBuild project is where the code bundle is built that will be pushed to Lambda. The default CloudFormation template will create a CodeBuild stage that builds a package using `chalice package` and then uploads those artifacts for CodePipeline to deploy.
- **S3 Buckets** - Two S3 buckets are created on your behalf.
  - **artifactbucketstore** - This bucket stores artifacts that are built by the CodeBuild project. The only artifact by default is the `transformed.yaml` created by the `aws cloudformation package` command.
  - **applicationbucket** - Stores the application bundle after the Chalice application has been packaged in the CodeBuild stage.
- Each resource is created with all the required IAM roles and policies.

CodeCommit repository

The CodeCommit repository can be added as a git remote for deployment. This makes it easy to kick off deployments. The developer doing the deployment only needs to push the release code up to the CodeCommit repository master branch. All the developer needs is keys that allow for push access to the CodeCommit repository. This is a lot easier than managing a set of `deployed.json` resources across a repsoitory and manually doing `chalice deploy` whenever a change needs to be deployed.

The default CodeCommit repository that is created is empty, you will have to populate it with the Chalice application code. Permissions will also need to be set up, you can find the documentation on how to do that here .

CodePipeline

CodePipeline is the main coordinator between all the other resources. It watches for changes on the CodeCommit repository, and triggers builds in the CodeBuild project. If the build succeeds then it will start a CloudFormation deployment of the built artifacts to a beta stage. This should be treated as a starting point, not a fully featured CD system.

CodeBuild build script

By default Chalice will create the CodeBuild project with a default buildspec that does the following.

```
version: 0.1
phases:
  install:
    commands:
    - sudo pip install --upgrade awscli
    - aws --version
    - sudo pip install chalice
    - sudo pip install -r requirements.txt
```

```
    - chalice package /tmp/packaged
    - aws cloudformation package --template-file
        tmp/packaged/sam.json --s3-bucket ${APP_S3_BUCKET}
        --output-template-file transformed.yaml
artifacts:
  type: zip
  files:
    - transformed.yaml
```

The CodeBuild stage installs both the AWS CLI and Chalice, then creates a package out of your chalice project, pushing the package to the application S3 bucket that was created for you. The transformed CloudFormation template is the only artifact, and can be run by CodePipeline after the build has succeeded.

Deploying to beta stage

Once the CodeBuild stage has finished building the Chalice package and creating the `transformed.yaml`, CodePipeline will take these artifacts and use them to create or update the beta stage. The `transformed.yaml` is a CloudFormation template that CodePipeline will execute, all the code it references has been uploaded to the application bucket by the AWS CLI in the CodeBuild stage, so this is the only artifact we need.

Once the CodePipeline beta build stage is finished, the beta version of the app is deployed and ready for testing.

Extending

It is recommended to use this pipeline as a starting point. The default template does not run any tests on the Chalice app before deploying to beta. There is also no mechanism provided by Chalice for a production stage. Ideally the CodeBuild stage would be used to run unit and functional tests before deploying to beta. After the beta stage is up, integration tests can be run against that endpoint, and if they all pass the beta stage could be promoted to a production stage using the CodePipleine manual approval feature.

CHAPTER 3

API Reference

Chalice

class Chalice (*app_name*)

This class represents a chalice application. It provides:

- The ability to register routes using the *route()* method.
- Within a view function, the ability to introspect the current request using the `current_request` attribute which is an instance of the *Request* class.

current_request

An object of type *Request*. This value is only set when a view function is being called. This attribute can be used to introspect the current HTTP request.

api

An object of type *APIGateway*. This attribute can be used to control how apigateway interprets `Content-Type` headers in both requests and responses.

lambda_context

A Lambda context object that is passed to the invoked view by AWS Lambda. You can find out more about this object by reading the lambda context object documentation.

debug

A boolean value that enables debugging. By default, this value is `False`. If debugging is true, then internal errors are returned back to the client. Additionally, debug log messages generated by the framework will show up in the cloudwatch logs. Example usage:

```
from chalice import Chalice

app = Chalice(app_name="appname")
app.debug = True
```

route (*path*, ***options*)

Register a view function for a particular URI path. This method is intended to be used as a decorator for a view function. For example:

```
from chalice import Chalice

app = Chalice(app_name="appname")

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

Parameters

- **path** (*str*) – The path to associate with the view function. The `path` should only contain `[a-zA-Z0-9._-]` chars and curly braces for parts of the URL you would like to capture. The path should not end in a trailing slash, otherwise a validation error will be raised during deployment.
- **methods** (*list*) – Optional parameter that indicates which HTTP methods this view function should accept. By default, only `GET` requests are supported. If you only wanted to support `POST` requests, you would specify `methods=['POST']`. If you support multiple HTTP methods in a single view function (`methods=['GET', 'POST']`), you can check the *app.current_request.method* attribute to see which HTTP method was used when making the request.
- **name** (*str*) – Optional parameter to specify the name of the view function. You generally do not need to set this value. The name of the view function is used as the default value for the view name.
- **authorizer** (*Authorizer*) – Specify an authorizer to use for this view. Can be an instance of *CognitoUserPoolAuthorizer*, *CustomAuthorizer* or *IAMAuthorizer*.
- **content_types** (*str*) – A list of content types to accept for this view. By default `application/json` is accepted. If this value is specified, then chalice will reject any incoming request that does not match the provided list of content types with a 415 Unsupported Media Type response.
- **api_key_required** (*boolean*) – Optional parameter to specify whether the method required a valid API key.
- **cors** – Specify if CORS is supported for this view. This can either by a boolean value or an instance of *CORSConfig*. Setting this value is set to `True` gives similar behavior to enabling CORS in the AWS Console. This includes injecting the `Access-Control-Allow-Origin` header to have a value of `*` as well as adding an `OPTIONS` method to support preflighting requests. If you would like more control over how CORS is configured, you can provide an instance of *CORSConfig*.

authorizer(*name*, ***options*)

Register a built-in authorizer.

```
from chalice import Chalice, AuthResponse

app = Chalice(app_name="appname")

@app.authorizer(ttl_seconds=30)
def my_auth(auth_request):
    # Validate auth_request.token, and then:
    return AuthResponse(routes=['/'], principal_id='username')

@app.route('/', authorizer=my_auth)
def viewfunction(value):
    pass
```

Parameters

- **ttl_seconds** – The number of seconds to cache this response. Subsequent requests that require this authorizer will use a cached response if available. The default is 300 seconds.

- **execution_role** – An optional IAM role to specify when invoking the Lambda function associated with the built-in authorizer.

schedule(*expression*, *name=None*)

Register a scheduled event that's invoked on a regular schedule. This will create a lambda function associated with the decorated function. It will also schedule the lambda function to be invoked with a scheduled CloudWatch Event.

See *Lambda Event Sources* for more information.

```
@app.schedule('cron(15 10 ? * 6L 2002-2005)')
def cron_handler(event):
    pass

@app.schedule('rate(5 minutes)')
def rate_handler(event):
    pass

@app.schedule(Rate(5, unit=Rate.MINUTES))
def rate_obj_handler(event):
    pass

@app.schedule(Cron(15, 10, '?', '*', '6L', '2002-2005'))
def cron_obj_handler(event):
    pass
```

Parameters

- **expression** – The schedule expression to use for the CloudWatch event rule. This value can either be a string value or an instance of type `ScheduleExpression`, which is either a *`Cron`* or *`Rate`* object. If a string value is provided, it will be provided directly as the `ScheduleExpression` value in the PutRule API call.
- **name** – The name of the function to use. This name is combined with the chalice app name as well as the stage name to create the entire lambda function name. This parameter is optional. If it is not provided, the name of the python function will be used.

lambda_function(*name=None*)

Create a pure lambda function that's not connected to anything.

See *Pure Lambda Functions* for more information.

Parameters **name** – The name of the function to use. This name is combined with the chalice app name as well as the stage name to create the entire lambda function name. This parameter is optional. If it is not provided, the name of the python function will be used.

Request

class Request

A class that represents the current request. This is mapped to the `app.current_request` object.

```
@app.route('/objects/{key}', methods=['GET', 'PUT'])
def myobject(key):
    request = app.current_request
    if request.method == 'PUT':
        # handle PUT request
        pass
```

```
    elif request.method == 'GET':
        # handle GET request
        pass
```

query_params
A dict of the query params for the request.

headers
A dict of the request headers.

uri_params
A dict of the captured URI params.

method
The HTTP method as a string.

json_body
The parsed JSON body (`json.loads(raw_body)`). This value will only be non-None if the Content-Type header is `application/json`, which is the default content type value in chalice.

raw_body
The raw HTTP body as bytes. This is useful if you need to calculate a checksum of the HTTP body.

context
A dict of additional context information.

stage_vars
A dict of configuration for the API Gateway stage.

to_dict()
Convert the *Request* object to a dictionary. This is useful for debugging purposes. This dictionary is guaranteed to be JSON serializable so you can return this value from a chalice view.

Response

class Response(*body*, *headers=None*, *status_code=200*)
A class that represents the response for the view function. You can optionally return an instance of this class from a view function if you want complete control over the returned HTTP response.

```
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'})
```

New in version 0.6.0.

body
The HTTP response body to send back. This value must be a string.

headers
An optional dictionary of HTTP headers to send back. This is a dictionary of header name to header value, e.g `{'Content-Type': 'text/plain'}`

status_code
The integer HTTP status code to send back in the HTTP response.

Authorization

Each of these classes below can be provided using the `authorizer` argument for an `@app.route(authorizer=...)` call:

```
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}
```

class CognitoUserPoolAuthorizer (*name*, *provider_arns*, *header='Authorization'*)
New in version 0.8.1.

name
The name of the authorizer.

provider_arns
The Cognito User Pool arns to use.

header
The header where the auth token will be specified.

class IAMAuthorizer
New in version 0.8.3.

class CustomAuthorizer (*name*, *authorizer_uri*, *ttl_seconds*, *header='Authorization'*)
New in version 0.8.1.

name
The name of the authorizer.

authorizer_uri
The URI of the lambda function to use for the custom authorizer. This usually has the form `arn:aws:apigateway:{region}:lambda:path/2015-03-01/functions/{lambda_arn}/invocatio`

ttl_seconds
The number of seconds to cache the returned policy from a custom authorizer.

header
The header where the auth token will be specified.

Built-in Authorizers

These classes are used when defining built-in authorizers in Chalice.

class AuthRequest (*auth_type*, *token*, *method_arn*)
An instance of this class is passed as the first argument to an authorizer defined via `@app.authorizer()`. You generally do not instantiate this class directly.

auth_type
The type of authentication

token
The authorization token. This is usually the value of the `Authorization` header.

method_arn
The ARN of the API gateway being authorized.

class AuthResponse (*routes*, *principal_id*, *context=None*)

routes
A list of authorized routes. Each element in the list can either by a string route such as *"/foo/bar"* or an instance of `AuthRoute`. If you specify the URL as a string, then all supported HTTP methods will be authorized. If you want to specify which HTTP methods are allowed, you can use `AuthRoute`. If you want to specify that all routes and HTTP methods are supported you can use the wildcard value of `"*"`: `AuthResponse(routes=['*'], ...)`

principal_id
The principal id of the user.

context
An optional dictionary of key value pairs. This dictionary will be accessible in the `app.current_request.context` in all subsequent authorized requests for this user.

class AuthRoute (*path*, *methods*)
This class be used in the `routes` attribute of a *`AuthResponse`* instance to get fine grained control over which HTTP methods are allowed for a given route.

path
The allowed route specified as a string

methods
A list of allowed HTTP methods.

APIGateway

class APIGateway
This class is used to control how API Gateway interprets `Content-Type` headers in both requests and responses.

There is a single instance of this class attached to each *`Chalice`* object under the `api` attribute.

default_binary_types
The value of `default_binary_types` are the `Content-Types` that are considered binary by default. This value should not be changed, instead you should modify the `binary_types` list to change the behavior of a content type. Its value is: `application/octet-stream`, `application/x-tar`, `application/zip`, `audio/basic`, `audio/ogg`, `audio/mp4`, `audio/mpeg`, `audio/wav`, `audio/webm`, `image/png`, `image/jpg`, `image/jpeg`, `image/gif`, `video/ogg`, `video/mpeg`, `video/webm`.

binary_types
The value of `binary_types` controls how API Gateway interprets requests and responses as detailed below.

If an incoming request has a `Content-Type` header value that is present in the `binary_types` list it will be assumed that its body is a sequence of raw bytes. You can access these bytes by accessing the `app.current_request.raw_body` property.

If an outgoing response from `Chalice` has a header `Content-Type` that matches one of the `binary_types` its body must be a `bytes` type object. It is important to note that originating request

must have the `Accept` header for the same type as the `Content-Type` on the response. Otherwise a `400` error will be returned.

This value can be modified to change what types API Gateway treats as binary. The easiest way to do this is to simply append new types to the list.

```
app.api.binary_types.append('application/my-binary-data')
```

Keep in mind that there can only be a total of 25 binary types at a time and Chalice by default has a list of 16 types. It is recommended if you are going to make extensive use of binary types to reset the list to the exact set of content types you will be using. This can easily be done by reassigning the whole list.

```
app.api.binary_types = [
    'application/octet-stream',
    'application/my-binary-data',
]
```

Implementation Note: API Gateway and Lambda communicate through a JSON event which is encoded using `UTF-8`. The raw bytes are temporarily encoded using base64 when being passed between API Gateway and Lambda. In the worst case this encoding can cause the binary body to be inflated up to `4/3` its original size. Lambda only accepts an event up to `6mb`, which means even if your binary data was not quite at that limit, with the base64 encoding it may exceed that limit. This will manifest as a `502` Bad Gateway error.

CORS

class CORSConfig(*allow_origin='*'*, *allow_headers=None*, *expose_headers=None*, *max_age=None*, *allow_credentials=None*)

CORS configuration to attach to a route.

```
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}
```

New in version 0.8.1.

allow_origin

The value of the `Access-Control-Allow-Origin` to send in the response. Keep in mind that 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.

allow_headers

The list of additional allowed headers. This list is added to list of built in allowed headers: `Content-Type`, `X-Amz-Date`, `Authorization`, `X-Api-Key`, `X-Amz-Security-Token`.

expose_headers
A list of values to return for the `Access-Control-Expose-Headers`:

max_age
The value for the `Access-Control-Max-Age`

allow_credentials
A boolean value that sets the value of `Access-Control-Allow-Credentials`.

Scheduled Events

New in version 1.0.0b1.

class Rate (*value*, *unit*)
An instance of this class can be used as the `expression` value in the *`Chalice.schedule()`* method:

```
@app.schedule(Rate(5, unit=Rate.MINUTES))
def handler(event):
    pass
```

Examples:

```
# Run every minute.
Rate(1, unit=Rate.MINUTES)

# Run every 2 hours.
Rate(2, unit=Rate.HOURS)
```

value
An integer value that presents the amount of time to wait between invocations of the scheduled event.

unit
The unit of the provided `value` attribute. This can be either `Rate.MINUTES`, `Rate.HOURS`, or `Rate.DAYS`.

MINUTES, HOURS, DAYS
These values should be used for the `unit` attribute.

class Cron (*minutes*, *hours*, *day_of_month*, *month*, *day_of_week*, *year*)
An instance of this class can be used as the `expression` value in the *`Chalice.schedule()`* method.

```
@app.schedule(Cron(15, 10, '?', '*', '6L', '2002-2005'))
def handler(event):
    pass
```

It provides more capabilities than the *`Rate`* class. There are a few limits:

- You can't specify `day_of_month` and `day_of_week` fields in the same Cron expression. If you specify a value in one of the fields, you must use a `?` in the other.
- Cron expressions that lead to rates faster than 1 minute are not supported.

For more information, see the API docs page.

Examples:

```
# Run at 10:00am (UTC) every day.
Cron(0, 10, '*', '*', '?', '*')

# Run at 12:15pm (UTC) every day.
```

```
    Cron(15, 12, '*', '*', '?', '*')

    # Run at 06:00pm (UTC) every Monday through Friday.
    Cron(0, 18, '?', '*', 'MON-FRI', '*')

    # Run at 08:00am (UTC) every 1st day of the month.
    Cron(0, 8, 1, '*', '?', '*')

    # Run every 15 minutes.
    Cron('0/15', '*', '*', '*', '?', '*')

    # Run every 10 minutes Monday through Friday.
    Cron('0/10', '*', '?', '*', 'MON-FRI', '*')

    # Run every 5 minutes Monday through Friday between
    # 08:00am and 5:55pm (UTC).
    Cron('0/5', '8-17', '?', '*', 'MON-FRI', '*')
```

class CloudWatchEvent

This is the input argument for a scheduled event.

```
    @app.schedule('rate(1 hour)')
    def every_hour(event: CloudWatchEvent):
        pass
```

In the code example above, the `event` argument is of type `CloudWatchEvent`, which will have the following attributes.

version
By default, this is set to 0 (zero) in all events.

account
The 12-digit number identifying an AWS account.

region
Identifies the AWS region where the event originated.

detail
For scheduled events, this will be an empty dictionary.

detail_type
For scheduled events, this value will be `"Scheduled Event"`.

source
Identifies the service that sourced the event. All events sourced from within AWS will begin with "aws." Customer-generated events can have any value here as long as it doesn't begin with "aws." We recommend the use of java package-name style reverse domain-name strings.

For scheduled events, this will be `aws.events`.

time
The event timestamp, which can be specified by the service originating the event. If the event spans a time interval, the service might choose to report the start time, so this value can be noticeably before the time the event is actually received.

event_id
A unique value is generated for every event. This can be helpful in tracing events as they move through rules to targets, and are processed.

resources
This JSON array contains ARNs that identify resources that are involved in the event. Inclusion of these

ARNs is at the discretion of the service.

For scheduled events, this will include the ARN of the CloudWatch rule that triggered this event.

to_dict()

Return the original event dictionary provided from Lambda. This is useful if you need direct access to the lambda event, for example if a new key is added to the lambda event that has not been mapped as an attribute to the `CloudWatchEvent` object. Example:

```
{'account': '123457940291',
 'detail': {},
 'detail-type': 'Scheduled Event',
 'id': '12345678-b9f1-4667-9c5e-39f98e9a6113',
 'region': 'us-west-2',
 'resources': ['arn:aws:events:us-west-2:123457940291:rule/testevents-dev-every_minute'],
 'source': 'aws.events',
 'time': '2017-06-30T23:28:38Z',
 'version': '0'}
```

CHAPTER 4

Upgrade Notes

Upgrade Notes

This document provides additional documentation on upgrading your version of chalice. If you're just interested in the high level changes, see the CHANGELOG.rst) file.

1.2.0

This release features a rewrite of the Chalice deployer (#604). This is a backwards compatible change, and should not have any noticeable changes with deployments with the exception of fixing deployer bugs (e.g. https://github.com/aws/chalice/issues/604). This code path affects the `chalice deploy`, `chalice delete`, and `chalice package` commands.

While this release is backwards compatible, you will notice several changes when you upgrade to version 1.2.0.

The output of `chalice deploy` has changed in order to give more details about the resources it creates along with a more detailed summary at the end:

```
$ chalice deploy
Creating deployment package.
Creating IAM role: myapp-dev
Creating lambda function: myapp-dev-foo
Creating lambda function: myapp-dev
Creating Rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:12345:function:myapp-dev-foo
  - Lambda ARN: arn:aws:lambda:us-west-2:12345:function:myapp-dev
  - Rest API URL: https://abcd.execute-api.us-west-2.amazonaws.com/api/
```

Also, the files used to store deployed values has changed. These files are used internally by the `chalice deploy/delete` commands and you typically do not interact with these files directly. It's mentioned here in case you notice new files in your `.chalice` directory. Note that these files are *not* part of the public interface of Chalice and are documented here for completeness and to help with debugging issues.

In versions < 1.2.0, the value of deployed resources was stored in `.chalice/deployed.json` and looked like this:

```
{
  "dev": {
    "region": "us-west-2",
    "api_handler_name": "demoauth4-dev",
    "api_handler_arn": "arn:aws:lambda:us-west-2:123:function:myapp-dev",
```

```
    "rest_api_id": "abcd",
    "lambda_functions": {
      "myapp-dev-foo": {
        "type": "pure_lambda",
        "arn": "arn:aws:lambda:us-west-2:123:function:myapp-dev-foo"
      }
    },
    "chalice_version": "1.1.1",
    "api_gateway_stage": "api",
    "backend": "api"
  },
  "prod": {...}
}
```

In version 1.2.0, the deployed resources are split into multiple files, one file per chalice stage. These files are in the `.chalice/deployed/<stage.json>`, so if you had a dev and a prod chalice stage you'd have `.chalice/deployed/dev.json` and `.chalice/deployed/prod.json`. The schema has also changed and looks like this:

```
$ cat .chalice/deployed/dev.json
{
  "schema_version": "2.0",
  "resources": [
    {
      "role_name": "myapp-dev",
      "role_arn": "arn:aws:iam::123:role/myapp-dev",
      "name": "default-role",
      "resource_type": "iam_role"
    },
    {
      "lambda_arn": "arn:aws:lambda:us-west-2:123:function:myapp-dev-foo",
      "name": "foo",
      "resource_type": "lambda_function"
    },
    {
      "lambda_arn": "arn:aws:lambda:us-west-2:123:function:myapp-dev",
      "name": "api_handler",
      "resource_type": "lambda_function"
    },
    {
      "name": "rest_api",
      "rest_api_id": "abcd",
      "rest_api_url": "https://abcd.execute-api.us-west-2.amazonaws.com/api",
      "resource_type": "rest_api"
    }
  ],
  "backend": "api"
}
```

When you run `chalice deploy` for the first time after upgrading to version 1.2.0, chalice will automatically converted `.chalice/deployed.json` over to the format as you deploy a given stage.

Warning: Once you upgrade to 1.2.0, chalice will only update the new `.chalice/deployed/<stage>.json`. This means you cannot downgrade to earlier versions of chalice unless you manually update `.chalice/deployed.json` as well.

The `chalice package` command has also been updated to use the deployer. This results in several changes

compared to the previous version:

- Pure lambdas are supported
- Scheduled events are supported
- Parity between the behavior of `chalice deploy` and `chalice package`

As part of this change, the CFN resource names have been updated to use `CamelCase` names. Previously, chalice converted your python function names to CFN resource names by removing all non alphanumeric characters and appending an md5 checksum, e.g `my_function -> myfunction3bfc`. With this new packager update, the resource name would be converted as `my_function -> MyFunction`. Note, the `Outputs` section renames unchanged in order to preserve backwards compatibility. In order to fix parity issues with `chalice deploy` and `chalice package`, we now explicitly create an IAM role resource as part of the default configuration.

1.0.0b2

The url parameter names and the function argument names must match. Previously, the routing code would use positional args `handler(*args)` to invoke a view function. In this version, kwargs are now used instead: `handler(**view_args)`. For example, this code will no longer work:

```
@app.route('/{a}/{b}')
def myview(first, second)
    return {}
```

The example above must be updated to:

```
@app.route('/{a}/{b}')
def myview(a, b)
    return {}
```

Now that functions are invoked with kwargs, the order doesn't matter. You may also write the above view function as:

```
@app.route('/{a}/{b}')
def myview(b, a)
    return {}
```

This was done to have consistent behavior with other web frameworks such as Flask.

1.0.0b1

The `Chalice.define_authorizer` method has been removed. This has been deprecated since v0.8.1. See *Authorization* for updated information on configuring authorizers in Chalice as well as the original deprecation notice in the *0.8.1* upgrade notes.

The optional deprecated positional parameter in the `chalice deploy` command for specifying the API Gateway stage has been removed. If you want to specify the API Gateway stage, you can use the `--api-gateway-stage` option in the `chalice deploy` command:

```
# Deprecated and removed in 1.0.0b1
$ chalice deploy prod

# Equivalent and updated way to specify an API Gateway stage:
$ chalice deploy --api-gateway-stage prod
```

0.9.0

The 0.9.0 release changed the type of `app.current_request.raw_body` to always be of type `bytes()`. This only affects users that were using python3. Previously you would get a type `str()`, but with the introduction of binary content type support, the `raw_body` attribute was made to consistently be of type `bytes()`.

0.8.1

The 0.8.1 changed the preferred way of specifying authorizers for view functions. You now specify either an instance of `chalice.CognitoUserPoolAuthorizer` or `chalice.CustomAuthorizer` to an `@app.route()` function using the `authorizer` argument.

Deprecated:

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

app.define_authorizer(
    name='MyPool',
    header='Authorization',
    auth_type='cognito_user_pools',
    provider_arns=['arn:aws:cognito:...:userpool/name']
)
```

Equivalent, and preferred way

```
from chalice import CognitoUserPoolAuthorizer

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}
```

The `define_authorizer` is still available, but is now deprecated and will be removed in future versions of chalice. You can also use the new `authorizer` argument to provider a `CustomAuthorizer`:

```
from chalice import CustomAuthorizer

authorizer = CustomAuthorizer(
    'MyCustomAuth', header='Authorization',
    authorizer_uri=('arn:aws:apigateway:region:lambda:path/2015-03-01'
                    '/functions/arn:aws:lambda:region:account-id:'
                    'function:FunctionName/invocations'))

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

0.7.0

The 0.7.0 release adds several major features to chalice. While the majority of these features are introduced in a backwards compatible way, there are a few backwards incompatible changes that were made in order to support these new major features.

Separate Stages

Prior to this version, chalice had a notion of a "stage" that corresponded to an API gateway stage. You can create and deploy a new API gateway stage by running `chalice deploy <stage-name>`. In 0.7.0, stage support was been reworked such that a chalice stage is a completely separate set of AWS resources. This means that if you have two chalice stages, say `dev` and `prod`, then you will have two separate sets of AWS resources, one set per stage:

- Two API Gateway Rest APIs
- Two separate Lambda functions
- Two separate IAM roles

The *Chalice Stages* doc has more details on the new chalice stages feature. This section highlights the key differences between the old stage behavior and the new chalice stage functionality in 0.7.0. In order to ease transition to this new model, the following changes were made:

- A new `--stage` argument was added to the `deploy`, `logs`, `url`, `generate-sdk`, and `package` commands. If this value is specified and the stage does not exist, a new chalice stage with that name will be created for you.
- The existing form `chalice deploy <stage-name>` has been deprecated. The command will still work in version 0.7.0, but a deprecation warning will be printed to stderr.
- If you want the pre-existing behavior of creating a new API gateway stage (while using the same Lambda function), you can use the `--api-gateway-stage` argument. This is the replacement for the deprecated form `chalice deploy <stage-name>`.
- The default stage if no `--stage` option is provided is `dev`. By defaulting to a `dev` stage, the pre-existing behavior of not specifying a stage name, e.g `chalice deploy`, `chalice url`, etc. will still work exactly the same.
- A new `stages` key is supported in the `.chalice/config.json`. This allows you to specify configuration specific to a chalice stage. See the *Configuration File* doc for more information about stage specific configuration.
- Setting `autogen_policy` to false will result in chalice looking for a IAM policy file named `.chalice/policy-<stage-name>.json`. Previously it would look for a file named `.chalice/policy.json`. You can also explicitly set this value to In order to ease transition, chalice will check for a `.chalice/policy.json` file when depoying to the `dev` stage. Support for `.chalice/policy.json` will be removed in future versions of chalice and users are encouraged to switch to the stage specific `.chalice/policy-<stage-name>.json` files.

See the *Chalice Stages* doc for more details on the new chalice stages feature.

Note, the AWS resource names it creates now have the form "<app-name>-<stage-name>", e.g. "myapp-dev", "myapp-prod".

We recommend using the new stage specific resource names. However, If you would like to use the existing resource names for a specific stage, you can create a `.chalice/deployed.json` file that specifies the existing values:

```
{
  "dev": {
    "backend": "api",
    "api_handler_arn": "lambda-function-arn",
    "api_handler_name": "lambda-function-name",
    "rest_api_id": "your-rest-api-id",
    "api_gateway_stage": "dev",
    "region": "your region (e.g us-west-2)",
    "chalice_version": "0.7.0",
```

```
    }
}
```

This file is discussed in the next section.

Deployed Values

In version 0.7.0, the way deployed values are stored and retrieved has changed. In prior versions, only the `lambda_arn` was saved, and its value was written to the `.chalice/config.json` file. Any of other deployed values that were needed (for example the API Gateway rest API id) was dynamically queried by assuming the resource names matches the app name. In this version of chalice, a separate `.chalice/deployed.json` file is written on every deployement which contains all the resources that have been created. While this should be a transparent change, you may noticed issues if you run commands such as `chalice url` and `chalice logs` without first deploying. To fix this issue, run `chalice deploy` and version 0.7.0 of chalice so a `.chalice/deployed.json` will be created for you.

Authorizer Changes

The ‘‘authorizer_id‘‘ and ‘‘authorization_type‘‘ args are no longer supported in ‘‘@app.route(...)‘‘ calls.

They have been replaced with an `authorizer_name` parameter and an `app.define_authorizer` method.

This version changed the internals of how an API gateway REST API is created. Prior to 0.7.0, the AWS SDK for Python was used to make the appropriate service API calls to API gateway include `create_rest_api` and `put_method / put_method_response` for each route. In version 0.7.0, this internal mechanism was changed to instead generate a swagger document. The rest api is then created or updated by calling `import_rest_api` or `put_rest_api` and providing the swagger document. This simplifies the internals and also unifies the code base for the newly added `chalice package` command (which uses a swagger document internally). One consequence of this change is that the entire REST API must be defined in the swagger document. With the previous `authorizer_id` parameter, you would create/deploy a rest api, create your authorizer, and then provide that `authorizer_id` in your `@app.route` calls. Now they must be defined all at once in the `app.py` file:

```
app = chalice.Chalice(app_name='demo')

@app.route('/auth-required', authorizer_name='MyUserPool')
def foo():
    return {}

app.define_authorizer(
    name='MyUserPool',
    header='Authorization',
    auth_type='cognito_user_pools',
    provider_arns=['arn:aws:cognito:...:userpool/name']
)
```

0.6.0

This version changed how the internals of how API gateway resources are created by chalice. The integration type changed from `AWS` to `AWS_PROXY`. This was to enable additional functionality, notable to allows users to provide non-JSON HTTP responses and inject arbitrary headers to the HTTP responses. While this change to the internals is primarily internal, there are several user-visible changes.

- Uncaught exceptions with `app.debug = False` (the default value) will result in a more generic `InternalServerError` error. The previous behavior was to return a `ChaliceViewError`.

- When you enabled debug mode via `app.debug = True`, the HTTP response will contain the python stack trace as the entire request body. This is to improve the readability of stack traces. For example:

```
$ http https://endpoint/dev/
HTTP/1.1 500 Internal Server Error
Content-Length: 358
Content-Type: text/plain

Traceback (most recent call last):
  File "/var/task/chalice/app.py", line 286, in __call__
    response = view_function(*function_args)
  File "/var/task/app.py", line 12, in index
    return a()
  File "/var/task/app.py", line 16, in a
    return b()
  File "/var/task/app.py", line 19, in b
    raise ValueError("Hello, error!")
ValueError: Hello, error!
```

- Content type validation now has error responses that match the same error response format used for other chalice built in responses. Chalice was previously relying on API gateway to perform the content type validation. As a result of the `AWS_PROXY` work, this logic has moved into the chalice handler and now has a consistent error response:

```
$ http https://endpoint/dev/ 'Content-Type: text/plain'
HTTP/1.1 415 Unsupported Media Type
Content-Type: application/json

{
    "Code": "UnsupportedMediaType",
    "Message": "Unsupported media type: text/plain"
}
```

- The keys in the `app.current_request.to_dict()` now match the casing used by the `AWS_PPROXY` lambda integration, which are `lowerCamelCased`. This method is primarily intended for introspection purposes.

Indices and tables

- genindex
- search

S

T

U

V