ChaliceWorkshop Documentation

Release 0.0.1

James Saryerwinnie

Jul 09, 2020

CONTENTS

Welcome to the AWS Chalice Workshop! This site contains various tutorials on how you can build serverless applications using AWS Chalice. To begin, please make sure your environment is set up correctly by following the Environment Setup tutorial. Then select one of the following tutorials to follow:

- Todo Application: A serverless web application to manage Todo's. This tutorial will walk through creating a serverless web API to create, update, get, and delete Todo's, managing Todo's in a database, adding authorization with JWT, and creating a full CI/CD pipeline for the application. AWS services covered include AWS Lambda, Amazon API Gateway, Amazon DynamoDB, AWS CodeBuild, and AWS CodePipeline.
- Media Query Application: A serverless application for querying media files in an Amazon S3 bucket. This tutorial will walk through using AWS Lambda event sources to create an automated workflow that processes uploaded media files and stores the processed information in a database. It will also walk through how to create a web API to query the processed information in the database. AWS services covered include AWS Lambda, Amazon Rekognition, Amazon S3, Amazon DynamoDB, Amazon API Gateway, and Amazon SNS.

CHAPTER

ONE

PREREQUISITE: SETTING UP YOUR ENVIRONMENT

To start working with AWS Chalice, there are some requirements your development environment must have:

- Python 3.7
- Virtualenv
- AWS credentials
- git

If you have all of the above requirements, you can skip these steps entirely.

1.1 Setting up Python

This workshop requires Python 3.7 for developing and running your Chalice application.

First, check to see if Python is already installed on your development environment:

```
$ python --version
Python 3.7.3
```

It is important to note that for this workshop, the version does not necessarily need to be `3.7.3`. The patch version can be any value as long as the major and minor version is `3.7`.

Installing Python will vary base on operating systems.

1.1.1 OS X

To install on OS X, make sure that `brew` is installed on your development environment:

```
$ brew --version
```

If `brew` is not installed (i.e. an error is thrown), then run the following command to install `brew`:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

With `brew` now installed, run to install Python:

```
$ brew install python
```

Once this command completes, check that Python now works on your machine by checking the Python version:

```
$ $(brew --prefix)/bin/python3 --version
Python 3.7.3
```

If a Python 3.7 version is returned, then you have successfully installed the required version of Python for this workshop.

1.1.2 Windows

To learn how to install Python on Windows, follow instructions from The Hitchhiker's Guide to Python

1.1.3 Linux

To learn how to install Python on Linux, follow instructions from The Hitchhiker's Guide to Python

1.2 Setting up AWS credentials

To use AWS Chalice, you will need AWS credentials. If you currently use one of the AWS SDKs or the AWS CLI on your development environment, you should already have AWS credentials set up and may skip this step. An easy way to check this is by checking that you have either a `~/.aws/credentials` or `~/.aws/config` file on your machine.

First if you do not have AWS account, create one on the sign up page.

To actually set up AWS credentials on your development environment, use the AWS CLI. To check if you have the AWS CLI installed, run:

```
$ aws --version
aws-cli/1.15.60 Python/3.7.3 Darwin/15.6.0 botocore/1.10.59
```

If it prints out a version, that means you have the AWS CLI installed on your development environment. To get credentials set, it should not matter what version of the AWS CLI you are using. The tutorial you choose to follow will inform you if you need a specific version of the AWS CLI.

If you do not have the AWS CLI v2 installed, you can install it by following the instructions in the user guide.

With the AWS CLI installed, run `aws configure` to configure your development environment for AWS credentials via its prompts:

```
$ aws configure
AWS Access Key ID [None]: ****************ABCD
AWS Secret Access Key [None]: ****************abCd
Default region name [None]: us-west-2
Default output format [None]:
```

For the `aws configure` command you will only need to provide an AWS Access Key ID, AWS Secret Access Key, and AWS region. To get an AWS Access Key and Secret Access Key, follow the instructions for creating these keys. For the AWS region, it is recommend to set this to `us-west-2`, but any region may be used.

Finally to check that everything is correctly set up, run the following AWS CLI:

```
$ aws ec2 describe-regions
```

This should return a JSON response back about all of the AWS regions supported by Amazon EC2. This indicates that the AWS credentials have been properly configured in your development environment.

1.3 Setting up git

You will need to clone a git repository so you should make sure you have have git installed on your development machine.

First, see if you already have git installed:

```
$ git --version
```

If you do not have git installed you will have to follow the section below for your system.

1.3.1 OS X

To install on OS X, make sure that `brew` is installed on your development environment:

```
$ brew --version
```

If `brew` is not installed (i.e. an error is thrown), then run the following command to install `brew`:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

With `brew` now installed, run to install git:

```
$ brew install git
```

1.3.2 Linux

Depending on your distro, git should be available in your standard package manager. Try one of the following commands:

```
$ sudo apt-get install git
```

```
$ sudo yum install git
```

1.3.3 Windows

For Windows, you will need to manually download and install a git client such as git-scm.

1.4 Optional requirements

Below is a set of tools that are not required to be installed but would facilitate the workshop:

1.4.1 Tree

A command line tool for recursively listing the structure of a directory. First check to see if you have `tree` installed:

```
$ tree --version
```

If it fails to return a version number, you should try to install it. To install on OSX, run the following:

```
$ brew install tree
```

For Linux, `tree` should be available in your standard package manager. Try one of the following commands:

```
$ sudo apt-get install tree
```

```
$ sudo yum install tree
```

CHAPTER

TWO

TODO APPLICATION

2.1 Part 1: Build a serverless web application with AWS Chalice

The first part of the workshop will introduce AWS Chalice and walk you through creating a Todo application using AWS Chalice.

2.1.1 Section 0: Introduction to AWS Chalice

This section will provide an introduction on how to use AWS Chalice and provide instructions on how to go about building your very first Chalice application.

Create a virtualenv and install Chalice

To start using Chalice, you will need a new virtualenv with Chalice installed.

Instructions

Make sure you have Python 3 installed. See the env-setup page for instructions on how to install Python.

1) Create a new virtualenv called `chalice-env` by running the following command:

```
$ python3 -m venv chalice-env
```

2) Activate your newly created virtualenv:

```
$ source chalice-env/bin/activate
```

If you are using a Windows environment, you will have to run:

```
> .\chalice-env\Scripts\activate
```

3) Install `chalice` using `pip`:

```
$ pip install chalice
```

Verification

To check that `chalice` was installed, run:

```
$ chalice --version
```

This should print out the version of `chalice` that is installed in your virtualenv.

Also, ensure that Python 3.7 is being used as the Python interpreter for your virtualenv:

```
$ python --version
Python 3.7.3
```

Create a new Chalice application

With `chalice` now installed, it is time to create your first Chalice application.

Instructions

1) Run the `chalice new-project` command to create a project called `workshop-intro`:

```
$ chalice new-project workshop-intro
```

Verification

A new `workshop-intro` directory should have been created on your behalf. Inside of the `workshop-intro` directory, you should have two files: an `app.py` file and a `requirements.txt` file:

```
$ ls workshop-intro
app.py           requirements.txt
```

Deploy the Chalice application

The newly created Chalice application can also be immediately deployed. So let's deploy it.

Instructions

1) Change directories to your newly created `workshop-intro` directory:

```
$ cd workshop-intro
```

2) Run `chalice deploy` to deploy your Chalice application:

```
$ chalice deploy
Creating deployment package.
Creating IAM role: workshop-intro-dev
Creating lambda function: workshop-intro-dev
Creating Rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:12345:function:workshop-intro-dev
  - Rest API URL: https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/
```

Verification

The `chalice deploy` command should have exited with a return code of `0`:

```
$ echo $?
0
```

You should also be able to interact with your newly deployed API. To do so, first install `httpie`:

```
$ pip install httpie
```

Get the endpoint of your deployed Chalice application with `chalice url`:

```
$ chalice url
https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/
```

Now use `httpie` to make an HTTP request to that endpoint:

```
$ http https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 18
Content-Type: application/json
Date: Sat, 21 Oct 2017 23:21:41 GMT
Via: 1.1 403d925786ea6bd8903b99a628977c8f.cloudfront.net (CloudFront)
X-Amz-Cf-Id: FlL4RfE3UqiDFocyTlSzCqtvzxWd9pK0M1lCnIsO1KwjhF37XvVTCg==
X-Amzn-Trace-Id: sampled=0;root=1-59ebd683-72e3a6105ff3425da0c7e0ae
X-Cache: Miss from cloudfront
x-amzn-RequestId: 9776fca3-b6b6-11e7-94e4-b130a115985d

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

The HTTP response back should consist of the JSON body: `{"hello": "world"}`

Add a new route

Now that we have deployed our first Chalice application, let's expand on it by adding a new `/hello` route.

Instructions

1) Open the `app.py` file in your favorite editor:

```
$ vim app.py
```

2) Inside of the `app.py` file, add the following function under the existing `index()` function:

```
@app.route('/hello')
def hello_workshop():
    return {'hello': 'workshop'}
```

Your `app.py` should now consist of the following:

```
from chalice import Chalice

app = Chalice(app_name='workshop-intro')


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

@app.route('/hello')
def hello_workshop():
    return {'hello': 'workshop'}
```

3) Deploy the updated application using `chalice deploy`:

```
$ chalice deploy
Creating deployment package.
Updating policy for IAM role: workshop-intro-dev
Updating lambda function: workshop-intro-dev
Updating rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:12345:function:workshop-intro-dev
  - Rest API URL: https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/
```

Validation

Using `httpie`, confirm that the new route was deployed by making an HTTP request:

```
$ http https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/hello
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 21
Content-Type: application/json
Date: Sat, 21 Oct 2017 23:34:56 GMT
Via: 1.1 2d8af5cc5befc5d35bb54b4a5b6494c9.cloudfront.net (CloudFront)
X-Amz-Cf-Id: upMVSIUvjmCRa33IO-4zpYQOU0C94h50F3oJX_iv-vdk-g1IacKq9A==
X-Amzn-Trace-Id: sampled=0;root=1-59ebd9a0-0a275c8f6794f2e5c59641c7
X-Cache: Miss from cloudfront
x-amzn-RequestId: 7233e21a-b6b8-11e7-a3b6-f7221d70ee14

{
    "hello": "workshop"
}
```

The HTTP response back should consist of the JSON body: `{"hello": "workshop"}`

Add a new route with a URI parameter

Next, let's add a new route that accepts a parameter in the URI.

Instructions

1) Inside of the `app.py` file, add the following function under the existing `hello_workshop()` function:

```
@app.route('/hello/{name}')
def hello_name(name):
    return {'hello': name}
```

Your `app.py` should now consist of the following:

```
from chalice import Chalice

app = Chalice(app_name='workshop-intro')


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

@app.route('/hello')
def hello_workshop():
    return {'hello': 'workshop'}

@app.route('/hello/{name}')
def hello_name(name):
    return {'hello': name}
```

2) Deploy the updated application using `chalice deploy`:

```
$ chalice deploy
Creating deployment package.
Updating policy for IAM role: workshop-intro-dev
Updating lambda function: workshop-intro-dev
Updating rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:12345:function:workshop-intro-dev
  - Rest API URL: https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/
```

Verification

Using `httpie`, confirm that the new route was deployed by making an HTTP request:

```
$ http https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/hello/kyle
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 21
Content-Type: application/json
Date: Sat, 21 Oct 2017 23:34:56 GMT
Via: 1.1 2d8af5cc5befc5d35bb54b4a5b6494c9.cloudfront.net (CloudFront)
X-Amz-Cf-Id: upMVSIUvjmCRa33IO-4zpYQOU0C94h50F3oJX_iv-vdk-g1IacKq9A==
```

(continues on next page)

(continued from previous page)

```
X-Amzn-Trace-Id: sampled=0;root=1-59ebd9a0-0a275c8f6794f2e5c59641c7
X-Cache: Miss from cloudfront
x-amzn-RequestId: 7233e21a-b6b8-11e7-a3b6-f7221d70ee14

{
    "hello": "kyle"
}
```

The HTTP response back should consist of the JSON body: `{"hello": "kyle"}`

Add a new route with a non-GET HTTP method

For our last route, let's add a new route that accepts a different HTTP method other than `GET`.

Instructions

1) Inside of the `app.py` file, add the following function under the existing `hello_name()` function:

```
@app.route('/hello-post', methods=['POST'])
def hello_post():
    request_body = app.current_request.json_body
    return {'hello': request_body}
```

Your `app.py` should now consist of the following:

```
from chalice import Chalice

app = Chalice(app_name='workshop-intro')


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

@app.route('/hello')
def hello_workshop():
    return {'hello': 'workshop'}

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

@app.route('/hello-post', methods=['POST'])
def hello_post():
    request_body = app.current_request.json_body
    return {'hello': request_body}
```

2) Deploy the updated application using `chalice deploy`:

```
$ chalice deploy
Creating deployment package.
Updating policy for IAM role: workshop-intro-dev
Updating lambda function: workshop-intro-dev
Updating rest API
```

(continues on next page)

(continued from previous page)

```
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:12345:function:workshop-intro-dev
  - Rest API URL: https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/
```

Verification

Using `httpie`, confirm that the new route was deployed by making an HTTP request:

```
$ echo '{"request":"body"}' | http POST https://1y2mueb824.execute-api.us-west-2.
↪amazonaws.com/api/hello-post
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 30
Content-Type: application/json
Date: Sat, 21 Oct 2017 23:48:43 GMT
Via: 1.1 805232684895bb3db77c2db44011c8d0.cloudfront.net (CloudFront)
X-Amz-Cf-Id: ah7w7to9Svn_WzGZ1MldMHERCO_sLxMKQi9AcHFLSjLtAdAPhw5z_A==
X-Amzn-Trace-Id: sampled=0;root=1-59ebdcdb-32c834bbd0341b40e3dfd787
X-Cache: Miss from cloudfront
x-amzn-RequestId: 5f0bf184-b6ba-11e7-a22d-9b7d2bcfb95b

{
    "hello": {
        "request": "body"
    }
}
```

Notice the HTTP response back should contain the JSON blob that was echoed into standard input.

Delete the Chalice application

Now with an understanding of the basics of how to use AWS Chalice, let's clean up this introduction application by deleting it remotely.

Instructions

1) Run `chalice delete` to delete the deployed AWS resources running this application:

```
$ chalice delete
Deleting Rest API: 1y2mueb824
Deleting function: arn:aws:lambda:us-west-2:12345:function:workshop-intro-dev
Deleting IAM role: workshop-intro-dev
```

   If you are prompted on whether to delete a resource when deleting the application, go ahead and confirm by entering `y`.

Verification

To ensure that the API no longer exists remotely, try to make an HTTP request to the endpoint it was originally deployed to:

```
$ http https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/

http: error: SSLError: [SSL: SSLV3_ALERT_HANDSHAKE_FAILURE] sslv3 alert
handshake failure (_ssl.c:590) while doing GET request to URL:
https://1y2mueb824.execute-api.us-west-2.amazonaws.com/api/
```

This should result in an SSL error as the remote application no longer exists and therefore it cannot be connected to it.

2.1.2 Section 1: Create initial Todo application

For the rest of this workshop, we will be building a serverless Todo application. The application will allow for creating Todo's, getting Todo's, updating Todo's, and deleting Todo's. In terms of the REST API, it will consist of the following:

HTTP Method	URI Path	Description
GET	/todos/	Gets a list of all Todo's
POST	/todos/	Creates a new Todo
GET	/todos/{id}	Gets a specific Todo
DELETE	/todos/{id}	Deletes a specific Todo
PUT	/todos/{id}	Updates the state of a Todo

Furthermore, a Todo will have the following schema:

```
{
  "description": {"type": "str"},
  "uid": {"type: "str"},
  "state": {"type: "str", "enum": ["unstarted", "started", "completed"]},
  "metadata": {
    "type": "object"
  },
  "username": {"type": "str"}
}
```

This step will focus on how to build a simple in-memory version of the Todo application. For this section we will be doing the following to create this version of the application:

- *Install Chalice*
- *Create a new Chalice project*
- *Add the starting `app.py`*
- *Add a route for creating a Todo*
- *Add a route for getting a specific Todo*
- *Add a route for deleting a specific Todo*
- *Add a route for updating the state of a specific Todo*
- *Final Code*

Install Chalice

This step will ensure that `chalice` is installed in your virtualenv.

Instructions

1. Install `chalice` inside of your virtualenv:

```
$ pip install chalice
```

Verification

To make sure `chalice` was installed correctly, run:

```
$ chalice --version
```

Create a new Chalice project

Create the new Chalice project for the Todo application.

Instructions

1. Create a new Chalice project called `mytodo` with the `new-project` command:

```
$ chalice new-project mytodo
```

Verification

To ensure that the project was created, list the contents of the newly created `mytodo` directory:

```
$ ls mytodo
app.py           requirements.txt
```

It should contain an `app.py` file and a `requirements.txt` file.

Add the starting `app.py`

Copy a boilerplate `app.py` file to begin working on the Todo application

Instructions

1. If you have not already done so, clone the repository for this workshop:

```
$ git clone https://github.com/aws-samples/chalice-workshop.git
```

2. Copy the over the `app.py` file to the `mytodo` Chalice application:

```
$ cp ../chalice-workshop/code/todo-app/part1/01-new-project/app.py mytodo/app.py
```

Verification

To verify that the boilerplate application is working correctly, move into the `mytodo` application directory and run `chalice local` to spin up a version of the application running locally:

```
$ cd mytodo
$ chalice local
Serving on localhost:8000
```

In a separate terminal window now install `httpie`:

```
$ pip install httpie
```

And make an HTTP request to application running the `localhost`:

```
$ http localhost:8000/todos
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:31:12 GMT
Server: BaseHTTP/0.3 Python/2.7.10

[]
```

This should return an empty list back as there are no Todo's currently in the application.

Add a route for creating a Todo

Add a route for creating a Todo.

Instructions

1. Open the `app.py` in an editor of your choice
2. At the bottom of the `app.py` file add a function called `add_new_todo()`
3. Decorate the `add_new_todo()` function with a `route` that only accepts `POST` to the URI `/todos`.
4. In the `add_new_todo()` function use the `app.current_request.json_body` to add the Todo (which includes its description and metadata) to the database.
5. In the `add_new_todo()` function `return` the ID of the Todo that was added in the database.

```
@app.route('/todos', methods=['POST'])
def add_new_todo():
    body = app.current_request.json_body
    return get_app_db().add_item(
        description=body['description'],
        metadata=body.get('metadata'),
    )
```

Verification

To verify that the new route works, run `chalice local` and in a separate terminal window run the following using `httpie`:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | http POST localhost:8000/
↪todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:44:24 GMT
Server: BaseHTTP/0.3 Python/2.7.10

8cc673f0-7dd3-4e9d-a20b-245fcd34859d
```

This will return the ID of the Todo. For this example, it is `8cc673f0-7dd3-4e9d-a20b-245fcd34859d`. Now check that it is now listed when you retrieve all Todos:

```
$ http localhost:8000/todos
HTTP/1.1 200 OK
Content-Length: 142
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:46:53 GMT
Server: BaseHTTP/0.3 Python/2.7.10

[
    {
        "description": "My first Todo",
        "metadata": {},
        "state": "unstarted",
        "uid": "8cc673f0-7dd3-4e9d-a20b-245fcd34859d",
        "username": "default"
    }
]
```

Add a route for getting a specific Todo

Add a route for getting a specific Todo.

Instructions

1. In the `app.py`, add a function called `get_todo()` that accepts a `uid` as a parameter.
2. Decorate the `get_todo()` function with a `route` that only accepts `GET` to the URI `/todos/{uid}`.
3. In the `get_todo()` function `return` the specific Todo item from the database using the `uid` function parameter.

```
@app.route('/todos/{uid}', methods=['GET'])
def get_todo(uid):
    return get_app_db().get_item(uid)
```

Verification

To verify that the new route works, run `chalice local` and in a separate terminal window run the following using `httpie`:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | http POST localhost:8000/
↪todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:44:24 GMT
Server: BaseHTTP/0.3 Python/2.7.10

8cc673f0-7dd3-4e9d-a20b-245fcd34859d
```

Now use the returned ID `8cc673f0-7dd3-4e9d-a20b-245fcd34859d` to request the specific Todo:

```
$ http localhost:8000/todos/8cc673f0-7dd3-4e9d-a20b-245fcd34859d
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:52:35 GMT
Server: BaseHTTP/0.3 Python/2.7.10

{
    "description": "My first Todo",
    "metadata": {},
    "state": "unstarted",
    "uid": "8cc673f0-7dd3-4e9d-a20b-245fcd34859d",
    "username": "default"
}
```

Add a route for deleting a specific Todo

Add a route for deleting a specific Todo.

Instructions

1. In the `app.py`, add a function called `delete_todo()` that accepts a `uid` as a parameter.
2. Decorate the `delete_todo()` function with a `route` that only accepts `DELETE` to the URI `/todos/{uid}`.
3. In the `delete_todo()` function delete the Todo from the database using the `uid` function parameter.

```
@app.route('/todos/{uid}', methods=['DELETE'])
def delete_todo(uid):
    return get_app_db().delete_item(uid)
```

Verification

To verify that the new route works, run `chalice local` and in a separate terminal window run the following using `httpie`:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | http POST localhost:8000/
↪todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:44:24 GMT
Server: BaseHTTP/0.3 Python/2.7.10

8cc673f0-7dd3-4e9d-a20b-245fcd34859d
```

Now check that it is now listed when you retrieve all Todos:

```
$ http localhost:8000/todos
HTTP/1.1 200 OK
Content-Length: 142
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:46:53 GMT
Server: BaseHTTP/0.3 Python/2.7.10

[
    {
        "description": "My first Todo",
        "metadata": {},
        "state": "unstarted",
        "uid": "8cc673f0-7dd3-4e9d-a20b-245fcd34859d",
        "username": "default"
    }
]
```

Now use the returned ID `8cc673f0-7dd3-4e9d-a20b-245fcd34859d` to delete the specific Todo:

```
$ http DELETE localhost:8000/todos/8cc673f0-7dd3-4e9d-a20b-245fcd34859d
HTTP/1.1 200 OK
```

(continues on next page)

(continued from previous page)

```
Content-Length: 4
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:57:32 GMT
Server: BaseHTTP/0.3 Python/2.7.10

null
```

Now if all of the Todo's are listed, it will no longer be present:

```
$ http localhost:8000/todos
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:31:12 GMT
Server: BaseHTTP/0.3 Python/2.7.10

[]
```

Add a route for updating the state of a specific Todo

Add a route for updating the state of a specific Todo.

Instructions

1. In the `app.py`, add a function called `update_todo()` that accepts a `uid` as a parameter.
2. Decorate the `update_todo()` function with a `route` that only accepts `PUT` to the URI `/todos/{uid}`.
3. In the `update_todo()` function use the `app.current_request` to update the Todo (which includes its description, metadata, and state) in the database for the `uid` provided.

```
@app.route('/todos/{uid}', methods=['PUT'])
def update_todo(uid):
    body = app.current_request.json_body
    get_app_db().update_item(
        uid,
        description=body.get('description'),
        state=body.get('state'),
        metadata=body.get('metadata'))
```

Verification

To verify that the new route works, run `chalice local` and in a separate terminal window run the following using `httpie`:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | http POST localhost:8000/
↪todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:44:24 GMT
Server: BaseHTTP/0.3 Python/2.7.10
```

(continues on next page)

(continued from previous page)

```
de9a4981-f7fd-4639-97fb-2af247f20d79
```

Now determine the state of this newly added Todo:

```
$ http localhost:8000/todos/de9a4981-f7fd-4639-97fb-2af247f20d79
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json
Date: Fri, 20 Oct 2017 00:03:26 GMT
Server: BaseHTTP/0.3 Python/2.7.10

{
    "description": "My first Todo",
    "metadata": {},
    "state": "unstarted",
    "uid": "de9a4981-f7fd-4639-97fb-2af247f20d79",
    "username": "default"
}
```

Update the `state` of this Todo to `started`:

```
$ echo '{"state": "started"}' | http PUT localhost:8000/todos/de9a4981-f7fd-4639-97fb-
↪2af247f20d79
HTTP/1.1 200 OK
Content-Length: 4
Content-Type: application/json
Date: Fri, 20 Oct 2017 00:05:07 GMT
Server: BaseHTTP/0.3 Python/2.7.10

null
```

Ensure that the Todo has the `started` state when described:

```
$ http localhost:8000/todos/de9a4981-f7fd-4639-97fb-2af247f20d79
HTTP/1.1 200 OK
Content-Length: 138
Content-Type: application/json
Date: Fri, 20 Oct 2017 00:05:54 GMT
Server: BaseHTTP/0.3 Python/2.7.10

{
    "description": "My first Todo",
    "metadata": {},
    "state": "started",
    "uid": "de9a4981-f7fd-4639-97fb-2af247f20d79",
    "username": "default"
}
```

Final Code

When you are done your final code should look like this:

```
from uuid import uuid4

from chalice import Chalice


app = Chalice(app_name='mytodo')
app.debug = True
_DB = None
DEFAULT_USERNAME = 'default'


class InMemoryTodoDB(object):
    def __init__(self, state=None):
        if state is None:
            state = {}
        self._state = state

    def list_all_items(self):
        all_items = []
        for username in self._state:
            all_items.extend(self.list_items(username))
        return all_items

    def list_items(self, username=DEFAULT_USERNAME):
        return self._state.get(username, {}).values()

    def add_item(self, description, metadata=None, username=DEFAULT_USERNAME):
        if username not in self._state:
            self._state[username] = {}
        uid = str(uuid4())
        self._state[username][uid] = {
            'uid': uid,
            'description': description,
            'state': 'unstarted',
            'metadata': metadata if metadata is not None else {},
            'username': username
        }
        return uid

    def get_item(self, uid, username=DEFAULT_USERNAME):
        return self._state[username][uid]

    def delete_item(self, uid, username=DEFAULT_USERNAME):
        del self._state[username][uid]

    def update_item(self, uid, description=None, state=None,
                    metadata=None, username=DEFAULT_USERNAME):
        item = self._state[username][uid]
        if description is not None:
            item['description'] = description
        if state is not None:
            item['state'] = state
        if metadata is not None:
            item['metadata'] = metadata
```

(continues on next page)

(continued from previous page)

```
55
56
57  def get_app_db():
58      global _DB
59      if _DB is None:
60          _DB = InMemoryTodoDB()
61      return _DB
62
63
64  @app.route('/todos', methods=['GET'])
65  def get_todos():
66      return get_app_db().list_items()
67
68
69  @app.route('/todos', methods=['POST'])
70  def add_new_todo():
71      body = app.current_request.json_body
72      return get_app_db().add_item(
73          description=body['description'],
74          metadata=body.get('metadata'),
75      )
76
77
78  @app.route('/todos/{uid}', methods=['GET'])
79  def get_todo(uid):
80      return get_app_db().get_item(uid)
81
82
83  @app.route('/todos/{uid}', methods=['DELETE'])
84  def delete_todo(uid):
85      return get_app_db().delete_item(uid)
86
87
88  @app.route('/todos/{uid}', methods=['PUT'])
89  def update_todo(uid):
90      body = app.current_request.json_body
91      get_app_db().update_item(
92          uid,
93          description=body.get('description'),
94          state=body.get('state'),
95          metadata=body.get('metadata'))
```

2.1.3 Section 2: Add `chalicelib` to Todo application

Users will learn about chalicelib in this section by moving the in-memory db out of `app.py` and into `chalicelib/db.py`

Our `app.py` file is getting a little bit crowded, and as our application grows it's only going to get worse. To solve this problem we can create a module called `chalicelib` that Chalice will deploy alongside the `app.py`

- *Create `chalicelib` module*
- *Move database code from `app.py` to the `db.py`*

Create `chalicelib` module

Let's start this process by moving our database code out of `app.py` and into `chalicelib`.

Instructions

1. Create a new `chalicelib` directory alongside the `app.py` file:

```
$ mkdir chalicelib
```

2. Since `chalicelib` is a Python module, it must have an `__init__.py` file:

```
$ touch chalicelib/__init__.py
```

3. Create a `db.py` file where all database interaction code will live:

```
$ touch chalicelib/db.py
```

Verification

The directory structure of your application should now look like this:

```
$ tree .
.
├── app.py
├── chalicelib
│   ├── __init__.py
│   └── db.py
└── requirements.txt

1 directory, 4 files
```

Move database code from `app.py` to the `db.py`

Copy `InMemoryTodoDB` class from `app.py` to `chalicelib/db.py`

Instructions

1. Cut the class `InMemoryTodoDB` out of `app.py` and paste it into `chalicelib/db.py` using your favorite editor
2. Move the following lines from `app.py` to `db.py`:

```
from uuid import uuid4


DEFAULT_USERNAME = 'default'
```

Verification

Lets try running `chalice local` and check a few routes to see if they still work:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | http POST localhost:8000/
↪todos
HTTP/1.1 500 Internal Server Error
Content-Length: 459
Content-Type: text/plain
Date: Fri, 20 Oct 2017 20:58:37 GMT
Server: BaseHTTP/0.3 Python/2.7.13

Traceback (most recent call last):
  File "/Users/jcarlyl/.envs/workshop/lib/python2.7/site-packages/chalice/app.py",␣
↪line 649, in _get_view_function_response
    response = view_function(**function_args)
  File "/private/tmp/chalice/add-db/app.py", line 24, in add_new_todo
    return get_app_db().add_item(
  File "/private/tmp/chalice/add-db/app.py", line 12, in get_app_db
    _DB = InMemoryTodoDB()
NameError: global name 'InMemoryTodoDB' is not defined
```

Since `InMemoryTodoDB` has been moved it now needs to be imported.

Import `InMemoryTodoDB` from chalicelib

Looks like we forgot to import the `InMemoryTodoDB` from `chalicelib`. Since `InMemoryTodoDB` is now in a different module, we need to import it.

Instructions

1. At the top of `app.py` add the line:

```
from chalicelib.db import InMemoryTodoDB
```

Verification

Let's try that last step one more time:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | \
    http POST localhost:8000/todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Fri, 20 Oct 2017 21:18:57 GMT
Server: BaseHTTP/0.3 Python/2.7.13

7fc955af-5a9e-42b5-ad3a-8f5017c91091
```

Now that it appears to work again let's finish verifying all the other routes still work as expected, starting with checking the state:

```
$ http localhost:8000/todos/7fc955af-5a9e-42b5-ad3a-8f5017c91091
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json
Date: Fri, 20 Oct 2017 21:21:03 GMT
Server: BaseHTTP/0.3 Python/2.7.13

{
    "description": "My first Todo",
    "metadata": {},
    "state": "unstarted",
    "uid": "7fc955af-5a9e-42b5-ad3a-8f5017c91091",
    "username": "default"
}
```

Update the `state` of this Todo to `started`:

```
$ echo '{"state": "started"}' | \
    http PUT localhost:8000/todos/7fc955af-5a9e-42b5-ad3a-8f5017c91091
HTTP/1.1 200 OK
Content-Length: 4
Content-Type: application/json
Date: Fri, 20 Oct 2017 21:21:59 GMT
Server: BaseHTTP/0.3 Python/2.7.13

null
```

Check the `state` again to make sure that it is now `started`:

```
$ http localhost:8000/todos/7fc955af-5a9e-42b5-ad3a-8f5017c91091
HTTP/1.1 200 OK
Content-Length: 138
Content-Type: application/json
Date: Fri, 20 Oct 2017 21:23:16 GMT
Server: BaseHTTP/0.3 Python/2.7.13

{
    "description": "My first Todo",
    "metadata": {},
    "state": "started",
    "uid": "7fc955af-5a9e-42b5-ad3a-8f5017c91091",
    "username": "default"
}
```

Final Code

When you are finished your `app.py` file should look like:

```
1  from chalice import Chalice
2  from chalicelib import db
3
4  app = Chalice(app_name='mytodo')
5  app.debug = True
6  _DB = None
7
8
```

(continues on next page)

(continued from previous page)

```
def get_app_db():
    global _DB
    if _DB is None:
        _DB = db.InMemoryTodoDB()
    return _DB


@app.route('/todos', methods=['GET'])
def get_todos():
    return get_app_db().list_items()


@app.route('/todos', methods=['POST'])
def add_new_todo():
    body = app.current_request.json_body
    return get_app_db().add_item(
        description=body['description'],
        metadata=body.get('metadata'),
    )


@app.route('/todos/{uid}', methods=['GET'])
def get_todo(uid):
    return get_app_db().get_item(uid)


@app.route('/todos/{uid}', methods=['DELETE'])
def delete_todo(uid):
    return get_app_db().delete_item(uid)


@app.route('/todos/{uid}', methods=['PUT'])
def update_todo(uid):
    body = app.current_request.json_body
    get_app_db().update_item(
        uid,
        description=body.get('description'),
        state=body.get('state'),
        metadata=body.get('metadata'))
```

And your `chalicelib/db.py` file should look like:

```
from uuid import uuid4


DEFAULT_USERNAME = 'default'


class InMemoryTodoDB(object):
    def __init__(self, state=None):
        if state is None:
            state = {}
        self._state = state

    def list_all_items(self):
        all_items = []
        for username in self._state:
```

(continues on next page)

(continued from previous page)

```
            all_items.extend(self.list_items(username))
        return all_items

    def list_items(self, username=DEFAULT_USERNAME):
        return self._state.get(username, {}).values()

    def add_item(self, description, metadata=None, username=DEFAULT_USERNAME):
        if username not in self._state:
            self._state[username] = {}
        uid = str(uuid4())
        self._state[username][uid] = {
            'uid': uid,
            'description': description,
            'state': 'unstarted',
            'metadata': metadata if metadata is not None else {},
            'username': username
        }
        return uid

    def get_item(self, uid, username=DEFAULT_USERNAME):
        return self._state[username][uid]

    def delete_item(self, uid, username=DEFAULT_USERNAME):
        del self._state[username][uid]

    def update_item(self, uid, description=None, state=None,
                    metadata=None, username=DEFAULT_USERNAME):
        item = self._state[username][uid]
        if description is not None:
            item['description'] = description
        if state is not None:
            item['state'] = state
        if metadata is not None:
            item['metadata'] = metadata
```

2.1.4 Section 3: Add a DynamoDB table for Todo application

In this step, we'll replace the in-memory database with an Amazon DynamoDB table.

Initial Setup

The starting code for this step is in the `chalice-workshop/code/todo-app/part1/03-add-dynamodb` file. If necessary, you can copy over those files as a starting point for this step:

```
$ cp ../chalice-workshop/code/todo-app/part1/03-add-dynamodb/app.py app.py
$ cp ../chalice-workshop/code/todo-app/part1/03-add-dynamodb/createtable.py␣
↪createtable.py
$ cp ../chalice-workshop/code/todo-app/part1/03-add-dynamodb/chalicelib/db.py␣
↪chalicelib/db.py
$ cp ../chalice-workshop/code/todo-app/part1/03-add-dynamodb/.chalice/policy-dev.json␣
↪.chalice/policy-dev.json
$ cp ../chalice-workshop/code/todo-app/part1/03-add-dynamodb/.chalice/config.json .
↪chalice/config.json
```

Create a DynamoDB table

In this section, we're going to create a DynamoDB table and configure chalice to pass in the table name to our application.

1. First, we'll need to install boto3, the AWS SDK for Python. Run this command:

```
$ pip install boto3
```

2. Add boto3 to our requirements.txt file. Chalice uses this file when building the deployment package for your app:

```
$ pip freeze | grep boto3 >> requirements.txt
```

3. Now that boto3 is installed, we can create the DynamoDB table Run the `createtable.py` script with the `--table-type app` option. This will take a few seconds to run.

```
$ python createtable.py --table-type app
```

5. Verify that this script added the table name to the `.chalice/config.json` file. You should see a key named `APP_TABLE_NAME` in this file:

```
$ cat .chalice/config.json
{
  "stages": {
    "dev": {
      "environment_variables": {
        "APP_TABLE_NAME": "todo-app-...."
      },
      "api_gateway_stage": "api"
    }
  },
  "version": "2.0",
  "app_name": "testapp"
}
```

6. Next, we'll add a test route to double check we've configured everything correctly. Open the `app.py` file and these import lines to the top of the file:

```
import os
import boto3
```

7. Add a new test route:

```
@app.route('/test-ddb')
def test_ddb():
    resource = boto3.resource('dynamodb')
    table = resource.Table(os.environ['APP_TABLE_NAME'])
    return table.name
```

Verification

1. Start up the local dev server: `chalice local`
2. Make a request to this test route and verify you get a 200 response:

```
$ http localhost:8000/test-ddb
HTTP/1.1 200 OK
Content-Length: 45
Content-Type: application/json
Server: BaseHTTP/0.3 Python/2.7.14

todo-app-0b116e7b-f0f8-4548-91d8-95c75898b8b6
```

Switching the `InMemoryTodoDB` to a `DynamoDBTodo`

Now that we've verified our DynamoDB table is plumbed into our chalice app correctly, we can update to use a new `DynamoDBTodo` backend instead of the `InMemoryTodoDB`.

The `chalicelib/db.py` file you copied from `code/todo-app/part1/03-add-dynamodb/chalicelib/db.py` has a new `DynamoDBTodo` class. This has the same interface as `InMemoryTodoDB` except that is uses DynamoDB as the backend. We're going to update our `app.py` to use this new class.

1. Remove the `@app.route('/test-ddb')` view function. We no longer need it now that we've verified that DynamoDB is correctly configured for our app.
2. Go to the `get_app_db()` function in your `app.py` file. Modify this function to use the `DynamoDBTodo` backend:

```
def get_app_db():
    global _DB
    if _DB is None:
        _DB = db.DynamoDBTodo(
            boto3.resource('dynamodb').Table(
                os.environ['APP_TABLE_NAME'])
        )
    return _DB
```

3. Go to the top of the `app.py` file. Modify the line `from chalicelib.db import InMemoryTodoDB` to reference `db` instead:

```
from chalicelib import db
```

Verification

1. Start up the local dev server `chalice local`
2. Create a Todo item:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | \
    http POST localhost:8000/todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Thu, 19 Oct 2017 23:44:24 GMT
```

(continues on next page)

(continued from previous page)

```
Server: BaseHTTP/0.3 Python/2.7.10

de9a4981-f7fd-4639-97fb-2af247f20d79
```

3. Retrieve the Todo item you just created. Keep in mind that your UID will be different from what's shown below:

```
$ http localhost:8000/todos/de9a4981-f7fd-4639-97fb-2af247f20d79
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json
Date: Fri, 20 Oct 2017 00:03:26 GMT
Server: BaseHTTP/0.3 Python/2.7.10

{
    "description": "My first Todo",
    "metadata": {},
    "state": "unstarted",
    "uid": "de9a4981-f7fd-4639-97fb-2af247f20d79",
    "username": "default"
}
```

Deploy your app

1. Now that we've tested locally, we're ready to deploy:

```
$ chalice deploy
```

Verification

1. First create a Todo item using the API Gateway endpoint:

```
$ chalice url
https://your-chalice-url/
$ echo '{"description": "My second Todo", "metadata": {}}' | \
    http POST https://your-chalice-url/todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json

abcdefg-abcdefg
```

2. Verify you can retrieve this item:

```
$ http https://your-chalice-url/todos/abcdefg-abcdefg
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json

{
    "description": "My second Todo",
    "metadata": {},
    "state": "unstarted",
    "uid": "abcdefg-abcdefg",
```

(continues on next page)

(continued from previous page)

```
    "username": "default"
}
```

2.1.5 Section 4: Add authorization to Todo application

If you had noticed from the previous steps, there was a `username` field for all of the Todos, but the `username` was always set to `default`. This step will be utilizing the `username` field by exposing the notion of users and authorization in the Todo application. For this section, we will be doing the following to add authorization and users to the application:

Install PyJWT

For authorization, the application is going to be relying on JWT. To depend on JWT, in the Chalice application `PyJWT` needs to be installed and added to our `requirements.txt` file.

Instructions

1) Add `PyJWT` to your `requirements.txt` file:

```
$ echo PyJWT==1.6.1 >> requirements.txt
```

2) Make sure it is now installed in your virtualenv:

```
$ pip install -r requirements.txt
```

Verification

To ensure that it was installed, open the Python REPL and try to import the `PyJWT` library:

```
$ python
  Python 2.7.10 (default, Mar 10 2016, 09:55:31)
  [GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
  Type "help", "copyright", "credits" or "license" for more information.
  >>> import jwt
```

Copy over auth specific files

In order to add authentication to your Chalice application we have provided a few files that help with some of the low-level details. We have added an `auth.py` file to `chalicelib` which abstracts away some of the details of handling JWT tokens. We have also added a `users.py` script which is a command line utility for creating and managing a user table.

Instructions

1) Copy in the `chalice-workshop/code/todo-app/part1/04-add-auth/chalicelib/auth.py` file:

```
$ cp ../chalice-workshop/code/todo-app/part1/04-add-auth/chalicelib/auth.py␣
↪chalicelib/auth.py
```

2) Copy over the `chalice-workshop/code/todo-app/part1/04-add-auth/users.py` script for creating users:

```
$ cp ../chalice-workshop/code/todo-app/part1/04-add-auth/users.py users.py
```

Verification

From within the `mytodo` directory of your Todo Chalice application, the structure should be the following:

```
$ tree
.
├── app.py
├── chalicelib
│   ├── __init__.py
│   ├── auth.py
│   └── db.py
├── createtable.py
├── requirements.txt
└── users.py
```

Create a DynamoDB user table

Using the `createtable.py` script, this will create another DynamoDB table for storing users to use in the Chalice application.

Instructions

1) Run the `createtable.py` script to create the DynamoDB table:

```
$ python createtable.py -t users
```

Verification

Check that the return code of the command is `0`:

```
$ echo $?
0
```

Also `cat` the `.chalice/config.json` to make sure the `USERS_TABLE_NAME` shows up as an environment variable:

```
$ cat .chalice/config.json
{
  "stages": {
    "dev": {
      "environment_variables": {
        "USERS_TABLE_NAME": "users-app-21658b12-517e-4441-baef-99b8fc2f0b61",
        "APP_TABLE_NAME": "todo-app-323ca4c3-54fb-4e49-a584-c52625e5d85d"
      },
      "autogen_policy": false,
      "api_gateway_stage": "api"
    }
  },
  "version": "2.0",
  "app_name": "mytodo"
}
```

Add a user to the user table

Using the `users.py` script, create a new user in your users database to use with your chalice application.

Instructions

1) Run the `users.py` script with the `-c` argument to create a user. You will be prompted for a username and a password:

```
$ python users.py -c
Username: user
Password:
```

Verification

Using the `users.py` script, make sure that the user is listed in your database:

```
$ python users.py -l
user
```

Also make sure that the password is correct by testing the username and password with the `users.py` script:

```
$ python users.py -t
Username: user
Password:
Password verified.
```

You can also test an incorrect password. You should see this output:

```
$ python users.py -t
Username: user
Password:
Password verification failed.
```

Create `get_users_db` function

Now that we have created a DynamoDB user table, we will create a convenience function for loading it.

Instructions

1. Add a new variable `_USER_DB` in your `app.py` file with a value of None:

```
app = Chalice(app_name='mytodo')
app.debug = True
_DB = None
# This is the new value you're adding.
_USER_DB = None
```

2. Create a function for fetching our current database table for users. Similar to the function that gets the app table. Add this function to your `app.py` file:

```
def get_users_db():
    global _USER_DB
    if _USER_DB is None:
        _USER_DB = boto3.resource('dynamodb').Table(
            os.environ['USERS_TABLE_NAME'])
    return _USER_DB
```

Create a login route

We will now create a login route where users can trade their username/password for a JWT token.

Instructions

1. Define a new Chalice route `/login` that accepts the POST method and grabs the `username` and `password` from the request, and forwards it along to a helper function in the `auth` code you copied in earlier which will trade those for a JWT token.

```
@app.route('/login', methods=['POST'])
def login():
    body = app.current_request.json_body
    record = get_users_db().get_item(
        Key={'username': body['username']})['Item']
    jwt_token = auth.get_jwt_token(
        body['username'], body['password'], record)
    return {'token': jwt_token}
```

2. Notice the above code snippit uses the `auth` file that we copied into our chalicelib directory at the beginning of this step. Add the following import statement to the top of `app.py` so we can use it:

```
from chalicelib import auth
```

Verification

1. Start up a local server using `chalice local`.
2. Using the username and password generated previously, run `chalice local` and make an HTTP `POST` request to the `/login` URI:

```
$ echo '{"username": "user", "password": "password"}' | \
    http POST localhost:8000/login
HTTP/1.1 200 OK
Content-Length: 218
Content-Type: application/json
Date: Fri, 20 Oct 2017 22:48:42 GMT
Server: BaseHTTP/0.3 Python/2.7.10

{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJpYXQiOjE1MDg1Mzk3MjIsImp0aSI6IjI5ZDJhNmFkLTdlY2YtNDYzZC1iOTY1LTk0M2VhNzU0YWMzYyIsInN1YiI6In
↪95hlpRWARK95aYCh0YE7ls_cvraoenNux8gmIy8vQU8"
}
```

This should return a JWT to use as an `Authorization` header for that user.

Create a custom authorizer and attach to a route

To add authorization to our app we will start by defining an authorizer and attaching it to one of our routes.

Instructions

1. Create an authorizer function that checks the validity of a JWT token using the existing code in the `auth.py` file we copied earlier. If the token is valid (didn't throw an error) we will return a policy that allows access to all of our routes, and sets the `principal_id` to the username in the JWT token.
2. Once we have defined the authorizer, we will attach it to the `get_todos` route.

```
@app.authorizer()
def jwt_auth(auth_request):
    token = auth_request.token
    decoded = auth.decode_jwt_token(token)
    return AuthResponse(routes=['*'], principal_id=decoded['sub'])
```

```
@app.route('/todos', methods=['GET'], authorizer=jwt_auth)
def get_todos():
```

Also make sure to import the `AuthResponse` class at the top of the `app.py` file:

```
from chalice import AuthResponse
```

Verification

1. Start the local dev server `chalice local`
2. Try to get the todo, the request should be rejected without authorization:

```
$ http localhost:8000/todos
HTTP/1.1 401 Unauthorized
Content-Length: 26
Content-Type: application/json
Date: Tue, 24 Oct 2017 02:50:50 GMT
Server: BaseHTTP/0.3 Python/2.7.13
x-amzn-ErrorType: UnauthorizedException
x-amzn-RequestId: 297d1da8-b9a8-4824-a1f3-293607aac715

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

3. Try the same call again but with your authorization token passed in the `Authorization` header:

```
$ http localhost:8000/todos \
    Authorization:eyJhbGciOi.... really long token here...
Content-Length: 137
Content-Type: application/json
Date: Tue, 24 Oct 2017 02:50:43 GMT
Server: BaseHTTP/0.3 Python/2.7.13

[
```

(continues on next page)

(continued from previous page)

```
    {
        "description": "My first Todo",
        "metadata": {},
        "state": "unstarted",
        "uid": "f9a992d6-41c0-45a6-84b8-e7239f7d7100",
        "username": "john"
    }
]
```

Attach authorizer to the rest of the routes

Now attach the authorizer to all the other routes except the `login` route.

Instructions

1. Attach the `jwt_auth` authorizer to the `add_new_todo` route.
2. Attach the `jwt_auth` authorizer to the `get_todo` route.
3. Attach the `jwt_auth` authorizer to the `delete_todo` route.
4. Attach the `jwt_auth` authorizer to the `update_todo` route.

```
@app.route('/todos', methods=['POST'], authorizer=jwt_auth)
def add_new_todo():
```

```
@app.route('/todos/{uid}', methods=['GET'], authorizer=jwt_auth)
def get_todo(uid):
```

```
@app.route('/todos/{uid}', methods=['DELETE'], authorizer=jwt_auth)
def delete_todo(uid):
```

```
@app.route('/todos/{uid}', methods=['PUT'], authorizer=jwt_auth)
def update_todo(uid):
```

Verification

1. Start up the local dev server `chalice local`
2. Try each route without an authorization token. You should get a `401` Unauthorized response:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | \
    http POST localhost:8000/todos
HTTP/1.1 401 Unauthorized
Content-Length: 26
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:14:14 GMT
Server: BaseHTTP/0.3 Python/2.7.13
x-amzn-ErrorType: UnauthorizedException
x-amzn-RequestId: 58c2d520-07e6-4535-b034-aaba41bab8ab

{
```

(continues on next page)

(continued from previous page)

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

```
$ http GET localhost:8000/todos/fake-id
HTTP/1.1 401 Unauthorized
Content-Length: 26
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:15:10 GMT
Server: BaseHTTP/0.3 Python/2.7.13
x-amzn-ErrorType: UnauthorizedException
x-amzn-RequestId: b2304a70-ff8d-453f-b119-10e75326463a

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

```
$ http DELETE localhost:8000/todos/fake-id
HTTP/1.1 401 Unauthorized
Content-Length: 26
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:17:10 GMT
Server: BaseHTTP/0.3 Python/2.7.13
x-amzn-ErrorType: UnauthorizedException
x-amzn-RequestId: 69419241-b244-462b-b108-72091f7d7b5b

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

```
$ echo '{"state": "started"}' | http PUT localhost:8000/todos/fake-id
HTTP/1.1 401 Unauthorized
Content-Length: 26
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:18:59 GMT
Server: BaseHTTP/0.3 Python/2.7.13
x-amzn-ErrorType: UnauthorizedException
x-amzn-RequestId: edc77f3d-3d3d-4a29-850a-502f21aeed96

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

3. Now try to create, get, update, and delete a todo from your application by using the `Authorization` header in all your requests:

```
$ echo '{"description": "My first Todo", "metadata": {}}' | \
     http POST localhost:8000/todos Authorization:eyJhbG... auth token ...
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:24:28 GMT
Server: BaseHTTP/0.3 Python/2.7.13

93dbabdb-3b2f-4029-845b-7754406c494f
```

```
$ echo '{"state": "started"}' | \
    http PUT localhost:8000/todos/93dbabdb-3b2f-4029-845b-7754406c494f \
    Authorization:eyJhbG... auth token ...
HTTP/1.1 200 OK
Content-Length: 4
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:25:28 GMT
Server: BaseHTTP/0.3 Python/2.7.13

null
```

```
$ http localhost:8000/todos/93dbabdb-3b2f-4029-845b-7754406c494f \
    Authorization:eyJhbG... auth token ...
HTTP/1.1 200 OK
Content-Length: 135
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:26:29 GMT
Server: BaseHTTP/0.3 Python/2.7.13

{
    "description": "My first Todo",
    "metadata": {},
    "state": "started",
    "uid": "93dbabdb-3b2f-4029-845b-7754406c494f",
    "username": "default"
}
```

```
$ http DELETE localhost:8000/todos/93dbabdb-3b2f-4029-845b-7754406c494f \
    Authorization:eyJhbG... auth token ...
HTTP/1.1 200 OK
Content-Length: 4
Content-Type: application/json
Date: Tue, 24 Oct 2017 03:27:10 GMT
Server: BaseHTTP/0.3 Python/2.7.13

null
```

Use authorizer provided username

Now that we have authorizers hooked up to all our routes we can use that instead of relying on the default user of `default`.

Instructions

1. First create a function named `get_authorized_username` that will be used to convert the information we have in our `current_request` into a username.

```
def get_authorized_username(current_request):
    return current_request.context['authorizer']['principalId']
```

2. Now we need to update each function that interacts with our database to calculate the `username` and pass it to the `xxx_item` method.

```
@app.route('/todos', methods=['GET'], authorizer=jwt_auth)
def get_todos():
    username = get_authorized_username(app.current_request)
    return get_app_db().list_items(username=username)


@app.route('/todos', methods=['POST'], authorizer=jwt_auth)
def add_new_todo():
    body = app.current_request.json_body
    username = get_authorized_username(app.current_request)
    return get_app_db().add_item(
        username=username,
        description=body['description'],
        metadata=body.get('metadata'),
    )


@app.route('/todos/{uid}', methods=['GET'], authorizer=jwt_auth)
def get_todo(uid):
    username = get_authorized_username(app.current_request)
    return get_app_db().get_item(uid, username=username)


@app.route('/todos/{uid}', methods=['DELETE'], authorizer=jwt_auth)
def delete_todo(uid):
    username = get_authorized_username(app.current_request)
    return get_app_db().delete_item(uid, username=username)


@app.route('/todos/{uid}', methods=['PUT'], authorizer=jwt_auth)
def update_todo(uid):
    body = app.current_request.json_body
    username = get_authorized_username(app.current_request)
    get_app_db().update_item(
        uid,
        description=body.get('description'),
        state=body.get('state'),
        metadata=body.get('metadata'),
        username=username)
```

Verification

1. Spin up the local Chalice server with `chalice local`.
2. Create a new todo and pass in your auth token:

```
$ echo '{"description": "a todo", "metadata": {}}' | \
     http POST localhost:8000/todos Authorization:eyJhbG... auth token ...
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json
Date: Tue, 24 Oct 2017 04:16:57 GMT
Server: BaseHTTP/0.3 Python/2.7.13

71048cc2-8583-41e5-9dfe-b9669d15af7d
```

3. List your todos using the get_todos route:

```
$ http localhost:8000/todos Authorization:eyJhbG... auth token ...
HTTP/1.1 200 OK
Content-Length: 132
Content-Type: application/json
Date: Tue, 24 Oct 2017 04:21:58 GMT
Server: BaseHTTP/0.3 Python/2.7.13

[
    {
        "description": "a todo",
        "metadata": {},
        "state": "unstarted",
        "uid": "7212a932-769b-4a19-9531-a950db7006a5",
        "username": "john"
    }
]
```

4. Notice that now the username is no longer `default` it should be whatever username went with the auth token you supplied.
5. Try making a new user with `python users.py -c` and then get their JWT token by calling the login route with their credentials.
6. Call the same route as above as the new user by passing in their JWT token in the `Authorization` header. They should get no todos since they have not created any yet:

```
http localhost:8000/todos 'Authorization:...the other auth token...'
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json
Date: Tue, 24 Oct 2017 04:25:56 GMT
Server: BaseHTTP/0.3 Python/2.7.13

[]
```

Deploying your authorizer code

Now that we have it working locally lets deploy it and verify that it still works.

Instructions

1. `chalice deploy` your app.

Verification

1. Try the same two calls above against the real API Gateway endpoint you get from your deploy instead of the localhost endpoint. If you lose your endpoint you can run `chalice url` which will print out your API Gateway endpoint:

```
$ http <your endpoint here>/todos \
    Authorization:...auth token that has no todos...
HTTP/1.1 200 OK
Connection: keep-alive
```

(continues on next page)

(continued from previous page)

```
Content-Length: 2
Content-Type: application/json
Date: Tue, 24 Oct 2017 04:43:20 GMT
Via: 1.1 cff9911a0035fa608bcaa4e9709161b3.cloudfront.net (CloudFront)
X-Amz-Cf-Id: bunfoZShHff_f3AqBPS2d5Ae3ymqgBusANDP9G6NvAZB3gOfr1IsVA==
X-Amzn-Trace-Id: sampled=0;root=1-59f01668-388cc9fa3db607662c2d623c
X-Cache: Miss from cloudfront
x-amzn-RequestId: 06de2818-b93f-11e7-bbb0-b760b41808da

[]
```

```
$ http <your endpoint here>/todos \
    Authorization:...auth token that has a todo...
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 132
Content-Type: application/json
Date: Tue, 24 Oct 2017 04:43:45 GMT
Via: 1.1 a05e153e17e2a6485edf7bf733e131a4.cloudfront.net (CloudFront)
X-Amz-Cf-Id: wR_7Bp4KglDjF41_9TNxXmc3Oiu2kll5XS1sTCCP_LD1kMC3C-nqOA==
X-Amzn-Trace-Id: sampled=0;root=1-59f01681-bb8ce2d74dc0c6f8fe095f9d
X-Cache: Miss from cloudfront
x-amzn-RequestId: 155f88f7-b93f-11e7-b351-775deacbeb7a

[
    {
        "description": "a todo",
        "metadata": {},
        "state": "unstarted",
        "uid": "7212a932-769b-4a19-9531-a950db7006a5",
        "username": "john"
    }
]
```

Final Code

When you are finished your `app.py` file should look like:

```
import os

import boto3
from chalice import Chalice, AuthResponse
from chalicelib import auth, db


app = Chalice(app_name='mytodo')
app.debug = True
_DB = None
_USER_DB = None


@app.route('/login', methods=['POST'])
def login():
    body = app.current_request.json_body
    record = get_users_db().get_item(
```

(continues on next page)

(continued from previous page)

```
        Key={'username': body['username']})['Item']
    jwt_token = auth.get_jwt_token(
        body['username'], body['password'], record)
    return {'token': jwt_token}


@app.authorizer()
def jwt_auth(auth_request):
    token = auth_request.token
    decoded = auth.decode_jwt_token(token)
    return AuthResponse(routes=['*'], principal_id=decoded['sub'])


def get_users_db():
    global _USER_DB
    if _USER_DB is None:
        _USER_DB = boto3.resource('dynamodb').Table(
            os.environ['USERS_TABLE_NAME'])
    return _USER_DB


# Rest API code


def get_app_db():
    global _DB
    if _DB is None:
        _DB = db.DynamoDBTodo(
            boto3.resource('dynamodb').Table(
                os.environ['APP_TABLE_NAME'])
        )
    return _DB


def get_authorized_username(current_request):
    return current_request.context['authorizer']['principalId']


@app.route('/todos', methods=['GET'], authorizer=jwt_auth)
def get_todos():
    username = get_authorized_username(app.current_request)
    return get_app_db().list_items(username=username)


@app.route('/todos', methods=['POST'], authorizer=jwt_auth)
def add_new_todo():
    body = app.current_request.json_body
    username = get_authorized_username(app.current_request)
    return get_app_db().add_item(
        username=username,
        description=body['description'],
        metadata=body.get('metadata'),
    )


@app.route('/todos/{uid}', methods=['GET'], authorizer=jwt_auth)
def get_todo(uid):
```

(continues on next page)

(continued from previous page)

```
    username = get_authorized_username(app.current_request)
    return get_app_db().get_item(uid, username=username)


@app.route('/todos/{uid}', methods=['DELETE'], authorizer=jwt_auth)
def delete_todo(uid):
    username = get_authorized_username(app.current_request)
    return get_app_db().delete_item(uid, username=username)


@app.route('/todos/{uid}', methods=['PUT'], authorizer=jwt_auth)
def update_todo(uid):
    body = app.current_request.json_body
    username = get_authorized_username(app.current_request)
    get_app_db().update_item(
        uid,
        description=body.get('description'),
        state=body.get('state'),
        metadata=body.get('metadata'),
        username=username)
```

2.2 Part 2: Deployment and setting up a CICD pipeline with AWS Chalice

The second part of the workshop will teach you how to productionize your AWS Chalice web application by walking you through how to set up a CICD pipeline for your application.

2.2.1 Section 1: `chalice package` command

In this section, we'll use the `chalice package` command to learn about the AWS CloudFormation integration with AWS chalice.

Initial Setup

We'll take our existing Todo app and create a SAM template.

Instructions

The starting code for this step is in `code/todo-app/part2/01-package-cmd`. You can reuse your existing sample application from part1 of this workshop. If necessary, you can copy over these files as a starting point for this section:

```
$ cp ../chalice-workshop/code/todo-app/part2/01-package-cmd/*.py .
$ cp ../chalice-workshop/code/todo-app/part2/01-package-cmd/chalicelib/*.py␣
↪chalicelib/
$ cp ../chalice-workshop/code/todo-app/part2/01-package-cmd/.chalice/policy-dev.json .
↪chalice/policy-dev.json
```

Now we're going to deploy our app using a CloudFormation stack.

1. First, ensure you have the AWS CLI installed.

```
$ aws --version
aws-cli/1.11.171 Python/2.7.14 Darwin/16.7.0 botocore/1.7.29
```

   If the AWS CLI is not installed, you can follow the instructions in the *Setting up AWS credentials* section.

Create a SAM template

In this step, we're going to create a SAM template using the `chalice package` command.

Instructions

1. Create a SAM template for your app by using the `chalice package` command:

```
$ chalice package packaged/
```

Verification

You should see two files in the `packaged/` directory, a deployment zip file as well as a SAM template.

1. Verify the contents of the `packaged/` directory:

```
$ ls -la packaged/
.
..
deployment.zip
sam.json
```

2. Verify the contents of the `deployment.zip`. You should see your app.py file along with all the python library dependencies needed to run your app. Chalice automatically handles managing dependencies based on your `requirements.txt` file:

```
$ unzip -l packaged/deployment.zip
Archive:  packaged/deployment.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
    31920  10-11-2017 16:28   chalice/app.py
      431  10-10-2017 11:40   chalice/__init__.py
      237  10-24-2017 11:30   app.py
             ...
     1159  10-24-2017 10:17   chalicelib/auth.py
     3647  10-24-2017 10:17   chalicelib/db.py
---------                     -------
```

3. Verify the contents of the `sam.json` file. You don't have to understand the specifics of this file, but you'll notice that there's a few serverless resources defined:

```
$ grep -B 1 'Serverless::' packaged/sam.json
    "RestAPI": {
      "Type": "AWS::Serverless::Api",
--
    "APIHandler": {
      "Type": "AWS::Serverless::Function",
```

Deploy your SAM template

Instructions

Next, we'll use the AWS CLI to deploy our application through AWS CloudFormation.

1. `cd` to the `packaged` directory:

```
$ cd packaged/
$ ls -la
.
..
deployment.zip
sam.json
```

2. Next you'll need to create an Amazon S3 bucket. When deploying your application with CloudFormation, your code is uploaded to an S3 bucket. We can use the AWS CLI to create an S3 bucket. Keep in mind that S3 buckets are globally unique, so you'll need to use your own bucket name:

```
$ aws s3 mb s3://chalice-workshop-cfn-bucket/ --region us-west-2
```

3. Use the AWS CLI to package your code. This will upload your code to the S3 bucket you've created and create a new SAM template that references your S3 object. Make sure to use the same bucket you used in the previous step for the value of the `--s3-bucket` option:

```
$ aws cloudformation package --template-file ./sam.json \
    --s3-bucket chalice-workshop-cfn-bucket \
    --output-template-file sam-packaged.yaml
```

4. Deploy your application using the AWS CLI.

```
$ aws cloudformation deploy --template-file ./sam-packaged.yaml \
    --stack-name chalice-beta-stack \
    --capabilities CAPABILITY_IAM
```

   This command will take a few minutes to execute. When this command finishes, you're chalice app will be up and running.

Verification

1. Verify that the stack creation was successful:

```
$ aws cloudformation describe-stacks --stack-name chalice-beta-stack \
    --query 'Stacks[0].StackStatus'
"CREATE_COMPLETE"
```

2. Query the stack outputs to retrieve the endpoint URL of your REST API:

```
$ aws cloudformation describe-stacks --stack-name chalice-beta-stack \
    --query 'Stacks[0].Outputs'
[
    {
        "OutputKey": "APIHandlerArn",
        "OutputValue": "arn:aws:lambda:us-west-2:123:function:..."
    },
```

(continues on next page)

(continued from previous page)

```
    {
        "OutputKey": "APIHandlerName",
        "OutputValue": "..."
    },
    {
        "OutputKey": "RestAPIId",
        "OutputValue": "abcd"
    },
    {
        "OutputKey": "EndpointURL",
        "OutputValue": "https://your-chalice-url/api/"
    }
]
```

3. Use the value for `EndpointURL` to test your API by creating a new Todo item:

```
$ echo '{"description": "My third Todo", "metadata": {}}' | \
    http POST https://your-chalice-url/api/todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json

abcdefg-abcdefg
```

4. Verify you can retrieve this item:

```
$ http https://your-chalice-url/todos/abcdefg-abcdefg
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json

{
    "description": "My third Todo",
    "metadata": {},
    "state": "unstarted",
    "uid": "abcdefg-abcdefg",
    "username": "default"
}
```

Update your app

Now we'll make a change and deploy our change.

Instructions

1. At the bottom of the `app.py` file, add a test route:

```
@app.route('/test-route', methods=['GET'])
def test_route():
    return {'test': 'route'}
```

2. Now we're going to use chalice and the AWS CLI to deploy this change. Make sure you're at the top level directory of your app (the app.py should be in your current working directory). Run the `chalice package` command:

```
$ ls -la
...
app.py
$ chalice package packaged/
```

3. Run the `aws cloudformation package` command. This will re-upload your code to S3. Be sure to use the same bucket name you used in the previous step:

```
$ cd packaged/
$ aws cloudformation package --template-file ./sam.json \
    --s3-bucket chalice-workshop-cfn-bucket \
    --output-template-file sam-packaged.yaml
```

4. Deploy your application using the AWS CLI:

```
$ aws cloudformation deploy --template-file ./sam-packaged.yaml \
    --stack-name chalice-beta-stack \
    --capabilities CAPABILITY_IAM
```

Verification

1. Verify that the stack update was successful:

```
$ aws cloudformation describe-stacks --stack-name chalice-beta-stack \
    --query 'Stacks[0].StackStatus'
```

2. Verify the new test route is available. Use the same `EndpointURL` from the previous step:

```
$ http https://your-chalice-url/api/test-route
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json

{"test": "route"}
```

Delete your stack

We no longer need this CloudFormation stack. In the next section we'll use AWS CodePipeline to manage this CloudFormation stack, so we can delete our existing stack. Rather that use `chalice delete`, we're going to use the AWS CLI to delete the CloudFormation stack we've created.

Instructions

1. Delete your CloudFormation stack:

```
$ aws cloudformation delete-stack --stack-name chalice-beta-stack
```

2. Wait for the deletion to successfully complete:

```
$ aws cloudformation wait stack-delete-complete \
    --stack-name chalice-beta-stack
```

3. Delete the S3 bucket you've created. Be sure to use the same bucket name you used when you created the bucket:

```
$ aws s3 rb --force s3://chalice-workshop-cfn-bucket/ \
    --region us-west-2
```

Verification

1. Verify the stack status:

```
$ aws cloudformation describe-stacks --stack-name chalice-beta-stack \
    --query 'Stacks[0].StackStatus'
```

2. Verify the `EndpointURL` is no longer accessible:

```
$ http https://your-chalice-url/api/test-route

http: error: SSLError: [SSL: SSLV3_ALERT_HANDSHAKE_FAILURE] sslv3 alert
handshake failure (_ssl.c:590) while doing GET request to URL:
https://your-chalice-url/api/test-route
```

2.2.2 Section 2: Working with AWS CodePipeline

In this section, we'll create a CodePipeline for our sample chalice app.

Creating a pipeline

AWS Chalice provides a command for generating a starter template. This template is managed through an AWS CloudFormation stack.

Instructions

1. Create a `release/` directory. We'll place CD related files in this directory:

```
$ mkdir release/
```

2. Generate a CloudFormation template for our starter CD pipeline:

```
$ chalice generate-pipeline release/pipeline.json
```

3. Deploy this template using the AWS CLI:

```
$ aws cloudformation deploy --stack-name chalice-pipeline-stack \
    --template-file release/pipeline.json \
    --capabilities CAPABILITY_IAM
```

This last command may take up a few minutes to deploy.

Configuring git

Up to this point, we have not been using any source control to track our changes to our sample app. We're now going to create and configure a git repo along with an AWS CodeCommit remote. If you haven't set up git, you can follow the instructions in the *Setting up git* section.

Instructions

1. Initialize your sample app as a git repository:

```
$ git init .
$ cp ../chalice-workshop/code/todo-app/part2/02-pipeline/.gitignore .
```

2. Commit your existing files:

```
$ git add -A .
$ git commit -m "Initial commit"
```

3. Query the CloudFormation stack you created in the previous step for the value of the remote repository:

```
$ aws cloudformation describe-stacks \
    --stack-name chalice-pipeline-stack \
    --query 'Stacks[0].Outputs'
[
    ...
    {
        "OutputKey": "SourceRepoURL",
        "OutputValue": "https://git-codecommit.us-west-2.amazonaws.com/v1/repos/
↪mytodo"
    },
    ...
]
```

4. Copy the value for the `SourceRepoURL` and configure a new git remote named `codecommit`. Be sure to use your value of the `SourceRepoURL`:

```
$ git remote add codecommit https://git-codecommit.us-west-2.amazonaws.com/v1/
↪repos/mytodo
```

5. Configure the CodeCommit credential helper. Append these lines to the end of your `.git/config` file:

```
[credential]
    helper =
    helper = !aws codecommit credential-helper $@
    UseHttpPath = true
```

Verification

1. Verify you have a `codecommit` remote:

```
$ git remote -v
codecommit  https://git-codecommit.us-west-2.amazonaws.com/v1/repos/mytodo (fetch)
codecommit  https://git-codecommit.us-west-2.amazonaws.com/v1/repos/mytodo (push)
```

2. Verify the credential helper is installed correctly. Mac users may see an `osxkeychain` entry as the first line of output. This is expected, you just need to verify the last two lines match the output below:

```
$ git config -l | grep helper
credential.helper=osxkeychain
credential.helper=
credential.helper=!aws codecommit credential-helper $@
```

3. Verify you can fetch from the `codecommit` remote:

```
$ git fetch codecommit
$ echo $?
0
```

Pushing your changes to AWS CodeCommit

Now we have our pipeline and git remote configured, anytime we push changes to our `codecommit` remote, our pipeline will automatically deploy our app.

Instructions

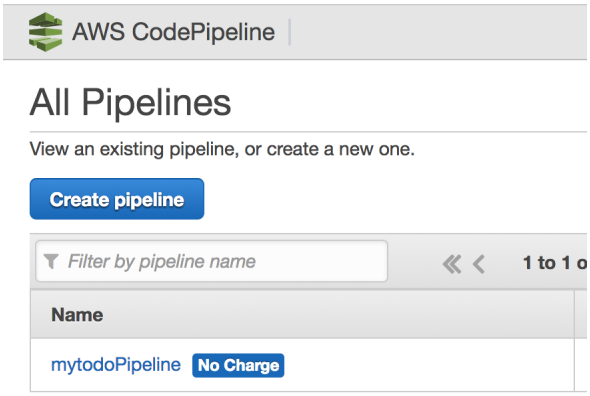
1. Push your changes to the `codecommit` remote:

```
$ git push codecommit master
Counting objects: 23, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (18/18), done.
Writing objects: 100% (23/23), 9.82 KiB | 3.27 MiB/s, done.
Total 23 (delta 2), reused 0 (delta 0)
To https://git-codecommit.us-west-2.amazonaws.com/v1/repos/mytodo
 * [new branch]      master -> master
```

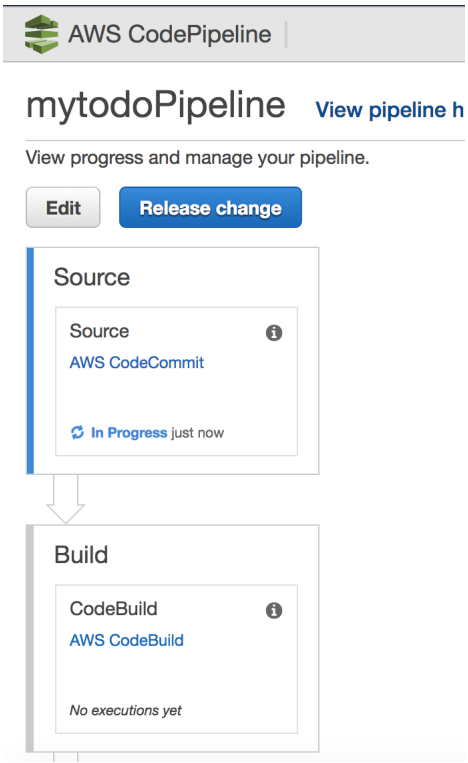
Verification

The best way to verify the pipeline is working as expected is to view the pipeline in the console:

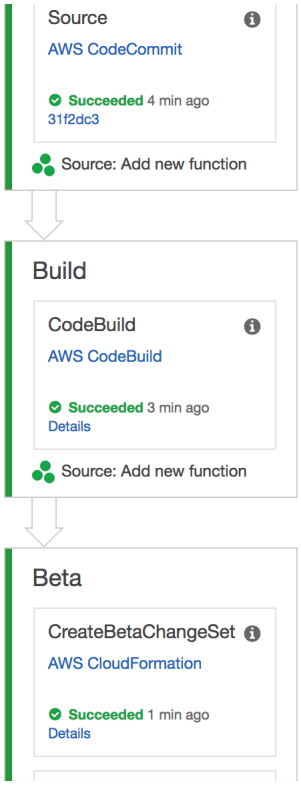
1. Log in to the AWS Console at https://console.aws.amazon.com/console/home
2. Go to the CodePipeline page.
3. Click on the "mytodoPipeline" pipeline.



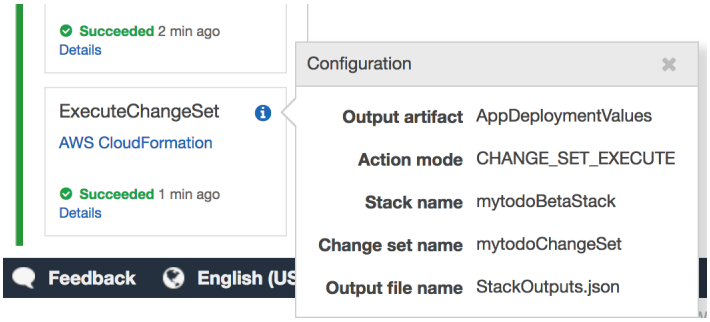
4. You should see a "Source", "Build", and "Beta" stage.
5. It can take a few minutes after pushing a change before the pipeline starts. If your pipeline has not started yet, wait a few minutes and refresh the page. Once the pipeline starts, it will take about 10 minutes for the intial deploy.



Wait until the stages have completed and all the stages are green.



6. Place your mouse over the "i" icon. Note the value of the **Stack name**. It should be something like `mytodoBetaStack`.



7. Query for the stack output of `EndpointURL` using the AWS CLI. This is the same step we performed in the previous section:

```
$ aws cloudformation describe-stacks --stack-name mytodoBetaStack \
    --query 'Stacks[0].Outputs'
[
    {
        "OutputKey": "APIHandlerArn",
        "OutputValue": "arn:aws:lambda:us-west-2:123:function:..."
    },
    {
        "OutputKey": "APIHandlerName",
        "OutputValue": "..."
    },
    {
        "OutputKey": "RestAPIId",
```

(continues on next page)

(continued from previous page)

```
        "OutputValue": "abcd"
    },
    {
        "OutputKey": "EndpointURL",
        "OutputValue": "https://your-chalice-url/api/"
    }
]
```

8. Use the value for `EndpointURL` to test your API by creating a new Todo item:

```
$ echo '{"description": "My third Todo", "metadata": {}}' | \
    http POST https://your-chalice-url/api/todos
HTTP/1.1 200 OK
Content-Length: 36
Content-Type: application/json

abcdefg-abcdefg
```

9. Verify you can retrieve this item:

```
$ http https://your-chalice-url/todos/abcdefg-abcdefg
HTTP/1.1 200 OK
Content-Length: 140
Content-Type: application/json

{
    "description": "My third Todo",
    "metadata": {},
    "state": "unstarted",
    "uid": "abcdefg-abcdefg",
    "username": "default"
}
```

Deploying an update

Now we'll make a change to our app and commit/push our change to CodeCommit. Our change will automatically be deployed.

Instructions

1. At the bottom of your `app.py` file, add a new test route:

```
@app.route('/test-pipeline')
def test_pipeline():
    return {'pipeline': 'route'}
```

2. Commit and push your changes:

```
$ git add app.py
$ git commit -m "Add test view"
$ git push codecommit master
Counting objects: 3, done.
Delta compression using up to 4 threads.
```

(continues on next page)

(continued from previous page)

```
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 357 bytes | 357.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://git-codecommit.us-west-2.amazonaws.com/v1/repos/mytodo
   4ded202..31f2dc3  master -> master
```

Verification

1. Go back to the AWS Console page for your CodePipeline named "mytodoPipeline".
2. Refresh the page. You should see the pipeline starting again. If you're not seeing any changes, you may need to wait a few minutes and refresh.
3. Wait for the pipeline to finish deploying.
4. Once it's finished verify the new test route is available. Use the same `EndpointURL` from the previous step:

```
$ http https://your-chalice-url/api/test-pipeline
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 21
Content-Type: application/json
...

{
    "pipeline": "route"
}
```

Extract the buildspec to a file

The instructions for how CodeBuild should package our app lives in the `release/pipeline.json` CloudFormation template. CodeBuild also supports loading the build instructions from a `buildspec.yml` file at the top level directory of your app. In this step we're going to extract out the build spec from the inline definition of the `release/pipeline.json` into a `buildspec.yml` file. This will allow us to modify how CodeBuild should build our app without having to redeploy our pipeline stack.

Instructions

1. Remove the `BuildSpec` key from your `release/pipeline.json` file. Your existing template has this section:

```
"Resources": {
  "AppPackageBuild": {
    "Type": "AWS::CodeBuild::Project",
      "Source": {
        "BuildSpec": " ... long string here ...",
        "Type": "CODEPIPELINE"
      }
    }
...
```

   And after removing the `BuildSpec` key it should look like this:

```
"Resources": {
  "AppPackageBuild": {
    "Type": "AWS::CodeBuild::Project",
      "Source": {
        "Type": "CODEPIPELINE"
      }
    }
...
```

2. Redeploying your pipeline stack using the AWS CLI:

```
$ aws cloudformation deploy --stack-name chalice-pipeline-stack \
    --template-file release/pipeline.json \
    --capabilities CAPABILITY_IAM
```

3. At the top level directory of your sample app, create a new file named `buildspec.yml` with these contents:

```
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
```

4. Commit the `buildspec.yml` file and push your changes to CodeCommit:

```
$ git add buildspec.yml
$ git commit -m "Adding buildspec.yml"
$ git push codecommit master
```

Verification

1. Go to the CodePipeline page in the console.
2. Wait for the pipeline to deploy your latest changes. Keep in mind that there should be no functional changes, we just want to verify that CodeBuild was able to load the `buildspec.yml` file.

Run unit tests

Now we're going to modify our `buildspec.yml` file to run our unit tests. If the tests fail our application won't deploy to our Beta stage.

Instructions

1. Create a new `build.sh` script with these contents:

```
#!/bin/bash
pip install --upgrade awscli
aws --version
pip install virtualenv
virtualenv /tmp/venv
. /tmp/venv/bin/activate
pip install -r requirements.txt
pip install -r requirements-test.txt
pip install chalice
export PYTHONPATH=.
py.test tests/ || exit 1
chalice package /tmp/packaged
aws cloudformation package --template-file /tmp/packaged/sam.json --s3-bucket "$
↪{APP_S3_BUCKET}" --output-template-file transformed.yaml
```

2. Make the script executable:

```
$ chmod +x ./build.sh
```

3. Update your `buildspec.yml` to call this build script:

```
version: 0.1
phases:
  install:
    commands:
      - sudo -E ./build.sh
artifacts:
  type: zip
  files:
    - transformed.yaml
```

4. Commit your changes and push them to codecommit:

```
$ git add build.sh buildspec.yml
$ git commit -m "Run unit tests"
```

Verification

1. Refresh your pipeline in the AWS console.
2. Verify the pipeline successfully completes.

Add a failing test

Now we'll add a failing unit test and verify that our application does not deploy.

Instructions

1. Add a failing test to the end of the `tests/test_db.py` file:

```
def test_fail():
    assert 0 == 1
```

2. Commit and push your changes:

```
$ git add tests/test_db.py
$ git commit -m "Add failing test"
$ git push codecommit master
```

Verification

1. Refresh your pipeline in the AWS console.
2. Verify that the CodeBuild stage fails.

CHAPTER

THREE

MEDIA QUERY APPLICATION

3.1 Part 0: Introduction to AWS Lambda and Chalice

This section will provide an introduction on how to use AWS Chalice and provide instructions on how to go about building your very first Chalice application running on AWS Lambda. Steps include:

3.1.1 Create a virtualenv and install Chalice

To start using Chalice, you will need a new virtualenv with Chalice installed.

Instructions

Make sure you have Python 3 installed. See the env-setup page for instructions on how to install Python.

1) Create a new virtualenv called `chalice-env` by running the following command:

```
$ python3 -m venv chalice-env
```

2) Activate your newly created virtualenv:

```
$ source chalice-env/bin/activate
```

If you are using a Windows environment, you will have to run:

```
> .\chalice-env\Scripts\activate
```

3) Install `chalice` using `pip`:

```
$ pip install chalice
```

Verification

1. To check that `chalice` was installed, run:

```
$ chalice --version
chalice 1.6.0, python 3.7.3, darwin 15.6.0
```

   The version of `chalice` must be version `1.6.0` or higher and the version of Python should be 3.7.

3.1.2 Create a new Chalice application

With `chalice` now installed, it is time to create your first Chalice application.

Instructions

1. Run the `chalice new-project` command to create a project called `workshop-intro`:

```
$ chalice new-project workshop-intro
```

Verification

1. A new `workshop-intro` directory should have been created on your behalf. Inside of the `workshop-intro` directory, you should have two files: an `app.py` file and a `requirements.txt` file:

```
$ ls workshop-intro
app.py           requirements.txt
```

3.1.3 Hello world Lambda function

Let's create our first Lambda function and deploy it using Chalice.

Instructions

1. Change directories to your newly created `workshop-intro` directory:

```
$ cd workshop-intro
```

2. Open the `app.py` file and delete **all** lines of code underneath the line: `app = Chalice(app_name='workshop-intro')`. Your `app.py` file should only consist of the following lines:

```
from chalice import Chalice

app = Chalice(app_name='workshop-intro')
```

3. Add a new function `hello_world` decorated by app.lambda_function() that returns `{"hello": "world"}`. Your `app.py` file should now consist of the following lines:

```
from chalice import Chalice

app = Chalice(app_name='workshop-intro')

@app.lambda_function()
def hello_world(event, context):
    return {'hello': 'world'}
```

4. Run `chalice deploy` to deploy your Chalice application to AWS Lambda:

```
$ chalice deploy
Creating deployment package.
Creating IAM role: workshop-intro-dev
Creating lambda function: workshop-intro-dev-hello_world
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:workshop-intro-dev-
↪hello_world
```

Verification

1. Run the `chalice invoke` command to invoke your newly deployed `hello_world` Lambda function:

```
$ chalice invoke -n hello_world
{"hello": "world"}
```

3.1.4 Lambda function using event parameter

Lambda functions accept two parameters: an `event` and a `context` parameter. The `event` parameter is used to provide data to the Lambda function. It is typically a dictionary, but may be a list, string, integer, float, or `None`. The `context` parameter provides information about the runtime to the Lambda function. This step will create a Lambda function that will use data from `event` passed to it to affect its return value.

Instructions

1. Create an additional Lambda function `hello_name` using the `app.lambda_function()` decorator. The function should retrieve the value of the `name` key in the `event` parameter and return `{'hello': name}`:

```
@app.lambda_function()
def hello_name(event, context):
    name = event['name']
    return {'hello': name}
```

Your `app.py` file should now consist of the following lines:

```
from chalice import Chalice

app = Chalice(app_name='workshop-intro')

@app.lambda_function()
def hello_world(event, context):
    return {'hello': 'world'}
```

(continues on next page)

(continued from previous page)

```
@app.lambda_function()
def hello_name(event, context):
    name = event['name']
    return {'hello': name}
```

2. Run `chalice deploy` to deploy your Chalice application with the new Lambda function:

```
$ chalice deploy
Creating deployment package.
Creating IAM role: workshop-intro-dev
Creating lambda function: workshop-intro-dev-hello_world
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:workshop-intro-dev-
↪hello_world
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:workshop-intro-dev-
↪hello_name
```

Verification

1. Run `chalice invoke` to invoke the `hello_name` Lambda function with `{"name": "Kyle"}` as the event payload:

```
$ echo '{"name": "Kyle"}' | chalice invoke -n hello_name
{"hello": "Kyle"}
```

2. It is also possible for your Lambda function to encounter runtime errors. Passing in an empty event payload when invoking the `hello_name` will result in the Lambda Function returning a Traceback:

```
$ chalice invoke -n hello_name
Traceback (most recent call last):
  File "/var/task/chalice/app.py", line 901, in __call__
    return self.func(event, context)
  File "/var/task/app.py", line 12, in hello_name
    name = event['name']
KeyError: 'name'
Error: Unhandled exception in Lambda function, details above.
```

3.1.5 Delete the Chalice application

Now with an understanding of the basics of AWS Lambda and Chalice, let's clean up this introduction application by deleting it remotely.

Instructions

1. Run `chalice delete` to delete the deployed Lambda functions running this application:

```
$ chalice delete
Deleting function: arn:aws:lambda:us-west-2:123456789123:function:workshop-intro-
↪dev-hello_name
Deleting function: arn:aws:lambda:us-west-2:123456789123:function:workshop-intro-
↪dev-hello_world
Deleting IAM role: workshop-intro-dev
```

Validation

1. Try running `chalice invoke` on the previously deployed Lambda functions:

```
$ chalice invoke -n hello_world
Could not find invokable resource with name: hello_world
$ chalice invoke -n hello_name
Could not find invokable resource with name: hello_name
```

   You should no longer be able to invoke both Lambda functions as they have been deleted.

3.2 Part 1: Introduction to Amazon Rekognition

The application being built will leverage Amazon Rekognition to detect objects in images and videos. This part of the tutorial will teach you more about Rekognition and how to detect objects with its API.

3.2.1 Install the AWS CLI

To interact with the Rekognition API, the AWS CLI will need to be installed.

Instructions

1. Check to see the CLI is installed:

```
$ aws --version
aws-cli/1.15.60 Python/3.6.5 Darwin/15.6.0 botocore/1.10.59
```

   The version of the CLI must be version 1.15.60 or greater. We recommend using AWS CLI v2.

2a. If the CLI is not installed, follow the installation instructions in the *Setting up AWS credentials* section.

2b. If your current CLI version is older than the minimum required version, follow the upgrade instructions in the user guide to upgrade to the latest version of the AWS CLI.

Verification

1. Run the following command:

```
$ aws --version
aws-cli/1.15.60 Python/3.6.1 Darwin/15.6.0 botocore/1.10.59
```

   The version displayed of the CLI must be version 1.15.60 or greater.

3.2.2 Detect image labels using Rekognition

Use the Rekognition API via the AWS CLI to detect labels in an image.

Instructions

1. If you have not already done so, clone the repository for this workshop:

```
$ git clone https://github.com/aws-samples/chalice-workshop.git
```

2. Use the `detect-labels` command to detect labels on a sample image:

```
$ aws rekognition detect-labels \
    --image-bytes fileb://chalice-workshop/code/media-query/final/assets/sample.
↪jpg
```

Verification

The output of the `detect-labels` command should be:

```
{
    "Labels": [
        {
            "Confidence": 85.75711822509766,
            "Name": "Animal"
        },
        {
            "Confidence": 85.75711822509766,
            "Name": "Canine"
        },
        {
            "Confidence": 85.75711822509766,
            "Name": "Dog"
        },
        {
            "Confidence": 85.75711822509766,
            "Name": "German Shepherd"
        },
        {
            "Confidence": 85.75711822509766,
            "Name": "Mammal"
        },
        {
            "Confidence": 85.75711822509766,
            "Name": "Pet"
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "Confidence": 84.56783294677734,
            "Name": "Collie"
        }
    ]
}
```

3.3 Part 2: Build a Chalice application using Rekognition

For this part of the tutorial, we will begin writing the media query Chalice application and integrate Rekognition into the application. This initial version of the application will accept the S3 bucket and key name of an image, call the `DetectLabels` API on that stored image, and return the labels detected for that image. So assuming the `sample.jpg` image is stored in a bucket `some-bucket` under the key `sample.jpg`, we will be able to invoke a Lambda function that return the labels Rekognition detected:

```
$ echo '{"Bucket": "some-bucket", "Key": "sample.jpg"}' | chalice invoke --name␣
↪detect_labels_on_image
["Animal", "Canine", "Dog", "German Shepherd", "Mammal", "Pet", "Collie"]
```

For this section, we will be doing the following to create this version of the application:

- *Create a new Chalice project*
- *Copy over boilerplate files*
- *Write a Lambda function for detecting labels*
- *Create a S3 bucket*
- *Deploy the Chalice application*

3.3.1 Create a new Chalice project

Create the new Chalice project for the Media Query application.

Instructions

1. Create a new Chalice project called `media-query` with the `new-project` command:

   ```
   $ chalice new-project media-query
   ```

Verification

To ensure that the project was created, list the contents of the newly created `media-query` directory:

```
$ ls media-query
app.py           requirements.txt
```

It should contain an `app.py` file and a `requirements.txt` file.

3.3.2 Copy over boilerplate files

Copy over starting files to facilitate development of the application

Instructions

1. Copy over the starting point code for section `02-chalice-with-rekognition` into your `media-query` directory:

```
$ cp -r chalice-workshop/code/media-query/02-chalice-with-rekognition/. media-
↪query/
```

Note: If you are ever stuck and want to skip to the beginning of a different part of this tutorial, you can do this by running the same command as above, but instead use the `code` directory name of the part you want to skip to. For example, if you wanted to skip to the beginning of Part 5 of this tutorial, you can run the following command with `media-query` as the current working directory and be ready to start Part 5:

```
media-query$  cp -r ../chalice-workshop/code/media-query/05-s3-delete-event/. ./
```

Verification

1. Ensure the structure of the `media-query` directory is the following:

```
$ tree -a media-query
├── .chalice
│   ├── config.json
│   └── policy-dev.json
├── .gitignore
├── app.py
├── chalicelib
│   ├── __init__.py
│   └── rekognition.py
├── recordresources.py
├── requirements.txt
└── resources.json
```

For the files that got added, they will be used later in the tutorial but for a brief overview of the new files:

- `chalicelib`: A directory for managing Python modules outside of the `app.py`. It is common to put the lower-level logic in the `chalicelib` directory and keep the higher level logic in the `app.py` file so it stays readable and small. You can read more about `chalicelib` in the Chalice documentation.
- `chalicelib/rekognition.py`: A utility module to further simplify `boto3` client calls to Amazon Rekognition.

- `.chalice/config.json`: Manages configuration of the Chalice application. You can read more about the configuration file in the Chalice documentation.
- `.chalice/policy-dev.json`: The IAM policy to apply to your Lambda function. This essentially manages the AWS permissions of your application
- `resources.json`: A CloudFormation template with additional resources to deploy outside of the Chalice application.
- `recordresources.py`: Records resource values from the additional resources deployed to your CloudFormation stack and saves them as environment variables in your Chalice application .

3.3.3 Write a Lambda function for detecting labels

Fill out the `app.py` file to write a Lambda function that detects labels on an image stored in a S3 bucket.

Instructions

1. Move into the `media-query` directory:

```
$ cd media-query
```

2. Add `boto3`, the AWS SDK for Python, as a dependency in the `requirements.txt` file:

```
boto3<1.8.0
```

3. Open the `app.py` file and delete all lines of code underneath the line: `app = Chalice(app_name='media-query')`. Your `app.py` file should only consist of the following lines:

```
from chalice import Chalice

app = Chalice(app_name='media-query')
```

3. Import `boto3` and the `chalicelib.rekognition` module in your `app.py` file:

```
import boto3
from chalice import Chalice
from chalicelib import rekognition
```

4. Add a helper function for instantiating a Rekognition client:

```
import boto3
from chalice import Chalice
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_REKOGNITION_CLIENT = None


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT
```

5. Add a new function `detect_labels_on_image` decorated by the `app.lambda_function` decorator. Have the function use a rekognition client to detect and return labels on an image stored in a S3 bucket:

```
import boto3
from chalice import Chalice
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_REKOGNITION_CLIENT = None


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.lambda_function()
def detect_labels_on_image(event, context):
    bucket = event['Bucket']
    key = event['Key']
    return get_rekognition_client().get_image_labels(bucket=bucket, key=key)
```

Verification

1. Ensure the contents of the `requirements.txt` file is:

```
boto3<1.8.0
```

1. Ensure the contents of the `app.py` file is:

```
import boto3
from chalice import Chalice
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_REKOGNITION_CLIENT = None


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.lambda_function()
def detect_labels_on_image(event, context):
    bucket = event['Bucket']
    key = event['Key']
    return get_rekognition_client().get_image_labels(bucket=bucket, key=key)
```

3.3.4 Create a S3 bucket

Create a S3 bucket for uploading images and use with the Chalice application.

Instructions

1. Use the AWS CLI and the `resources.json` CloudFormation template to deploy a CloudFormation stack `media-query` that contains a S3 bucket:

```
$ aws cloudformation deploy --template-file resources.json --stack-name media-
↪query
```

Verification

1. Retrieve and store the name of the S3 bucket using the AWS CLI:

```
$ MEDIA_BUCKET_NAME=$(aws cloudformation describe-stacks --stack-name media-query␣
↪--query "Stacks[0].Outputs[?OutputKey=='MediaBucketName'].OutputValue" --output␣
↪text)
```

2. Ensure you can access the S3 bucket by listing its contents:

```
$ aws s3 ls $MEDIA_BUCKET_NAME
```

   Note that the bucket should be empty.

3.3.5 Deploy the Chalice application

Deploy the chalice application.

Instructions

1. Install the dependencies of the Chalice application:

```
$ pip install -r requirements.txt
```

2. Run `chalice deploy` to deploy the application:

```
$ chalice deploy
Creating deployment package.
Creating IAM role: media-query-dev-detect_labels_on_image
Creating lambda function: media-query-dev-detect_labels_on_image
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪detect_labels_on_image
```

Verification

1. Upload the sample workshop image to the S3 bucket:

```
$ aws s3 cp ../chalice-workshop/code/media-query/final/assets/sample.jpg s3://
↪$MEDIA_BUCKET_NAME
```

2. Create a `sample-event.json` file to use with `chalice invoke`:

```
$ echo "{\"Bucket\": \"$MEDIA_BUCKET_NAME\", \"Key\": \"sample.jpg\"}" > sample-
↪event.json
```

3. Run `chalice invoke` on the `detect_labels_on_image` Lambda function:

```
$ chalice invoke --name detect_labels_on_image < sample-event.json
```

It should return the following labels in the output:

```
["Animal", "Canine", "Dog", "German Shepherd", "Mammal", "Pet", "Collie"]
```

3.4 Part 3: Integrate with a DynamoDB table

Now that we have a Lambda function that can detect labels in an image, let's integrate a DynamoDB table so we can query information across the various images stored in our bucket. So instead of returning the labels, the Chalice application will store the items in a DynamoDB table.

For this section, we will be doing the following to integrate the DynamoDB table:

3.4.1 Copy over boilerplate files

Copy over files needed for integrating the DynamoDB table into the application

Instructions

1. Using `media-query` as the current working directory, copy the `db.py` module into the `chalicelib` package:

```
$ cp ../chalice-workshop/code/media-query/03-add-db/chalicelib/db.py chalicelib/
```

2. Using `media-query` as the current working directory, copy over an updated version of the `resources.json` file:

```
$ cp ../chalice-workshop/code/media-query/03-add-db/resources.json .
```

Verification

1. Ensure the structure of the `media-query` directory includes the following files and directories:

```
$ tree -a .
├── .chalice
│   ├── config.json
│   └── policy-dev.json
├── .gitignore
├── app.py
├── chalicelib
│   ├── __init__.py
│   ├── db.py
│   └── rekognition.py
├── recordresources.py
├── requirements.txt
└── resources.json
```

   Note there will be more files listed with `tree` assuming you already deployed the application once. However, the files listed from the `tree` output above are required.

2. Ensure the contents of the `resources.json` is now the following:

```
$ cat resources.json
{
  "Outputs": {
    "MediaBucketName": {
      "Value": {
        "Ref": "MediaBucket"
      }
    },
    "MediaTableName": {
      "Value": {
        "Ref": "MediaTable"
      }
    }
  },
  "Resources": {
    "MediaBucket": {
      "Type": "AWS::S3::Bucket"
    },
    "MediaTable": {
      "Properties": {
        "AttributeDefinitions": [
          {
            "AttributeName": "name",
            "AttributeType": "S"
          }
        ],
        "KeySchema": [
          {
            "AttributeName": "name",
            "KeyType": "HASH"
          }
        ],
        "ProvisionedThroughput": {
          "ReadCapacityUnits": 5,
          "WriteCapacityUnits": 5
```

(continues on next page)

(continued from previous page)

```
        }
      },
      "Type": "AWS::DynamoDB::Table"
    }
  }
}
```

3.4.2 Create a DynamoDB table

Create a DynamoDB table to store and query information about images in the S3 bucket.

Instructions

1. Use the AWS CLI and the `resources.json` CloudFormation template to redeploy the `media-query` CloudFormation stack and create a new DynamoDB

```
$ aws cloudformation deploy --template-file resources.json --stack-name media-
↪query
```

Verification

1. Retrieve and store the name of the DynamoDB table using the AWS CLI:

```
$ MEDIA_TABLE_NAME=$(aws cloudformation describe-stacks --stack-name media-query -
↪-query "Stacks[0].Outputs[?OutputKey=='MediaTableName'].OutputValue" --output␣
↪text)
```

2. Ensure the existence of the table using the `describe-table` CLI command:

```
$ aws dynamodb describe-table --table-name $MEDIA_TABLE_NAME
{
    "Table": {
        "AttributeDefinitions": [
            {
                "AttributeName": "name",
                "AttributeType": "S"
            }
        ],
        "TableName": "media-query-MediaTable-10QEPR0O8DOT4",
        "KeySchema": [
            {
                "AttributeName": "name",
                "KeyType": "HASH"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": 1531769158.804,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
```

(continues on next page)

(continued from previous page)

```
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-west-2:123456789123:table/media-query-
↪MediaTable-10QEPR0O8DOT4",
        "TableId": "00eebe92-d59d-40a2-b5fa-32e16b571cdc"
    }
}
```

3.4.3 Integrate the DynamoDB table

Integrate the newly created DynamoDB table into the Chalice application.

Instructions

1. Save the DynamoDB table name as an environment variable in the Chalice application by running the `recordresources.py` script:

```
$ python recordresources.py --stack-name media-query
```

2. Import `os` and the `chalicelib.db` module in your `app.py` file:

```
import os

import boto3
from chalice import Chalice
from chalicelib import db
from chalicelib import rekognition
```

3. Add a helper function for instantiating a `db.DynamoMediaDB` class using the DynamoDB table name stored as the environment variable `MEDIA_TABLE_NAME`:

```
import os

import boto3
from chalice import Chalice
from chalicelib import db
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_MEDIA_DB = None
_REKOGNITION_CLIENT = None


def get_media_db():
    global _MEDIA_DB
    if _MEDIA_DB is None:
        _MEDIA_DB = db.DynamoMediaDB(
            boto3.resource('dynamodb').Table(
                os.environ['MEDIA_TABLE_NAME']))
    return _MEDIA_DB
```

4. Update the `detect_labels_on_image` Lambda function to save the image along with the detected labels to the database:

```
@app.lambda_function()
def detect_labels_on_image(event, context):
    bucket = event['Bucket']
    key = event['Key']
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)
```

Verification

1. Ensure the contents of the `config.json` contains environment variables for `MEDIA_TABLE_NAME`:

```
$ cat .chalice/config.json
{
  "version": "2.0",
  "app_name": "media-query",
  "stages": {
    "dev": {
      "api_gateway_stage": "api",
      "autogen_policy": false,
      "environment_variables": {
        "MEDIA_TABLE_NAME": "media-query-MediaTable-10QEPR0O8DOT4",
        "MEDIA_BUCKET_NAME": "media-query-mediabucket-fb8oddjbslv1"
      }
    }
  }
}
```

   Note that the `MEDIA_BUCKET_NAME` will be present as well in the environment variables. It will be used in the next part of the tutorial.

2. Ensure the contents of the `app.py` file is:

```
import os

import boto3
from chalice import Chalice
from chalicelib import db
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_MEDIA_DB = None
_REKOGNITION_CLIENT = None


def get_media_db():
    global _MEDIA_DB
    if _MEDIA_DB is None:
        _MEDIA_DB = db.DynamoMediaDB(
            boto3.resource('dynamodb').Table(
                os.environ['MEDIA_TABLE_NAME']))
    return _MEDIA_DB


def get_rekognition_client():
    global _REKOGNITION_CLIENT
```

(continues on next page)

(continued from previous page)

```
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.lambda_function()
def detect_labels_on_image(event, context):
    bucket = event['Bucket']
    key = event['Key']
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)
```

3.4.4 Redeploy the Chalice application

Deploy the updated Chalice application.

Instructions

1. Run `chalice deploy`:

```
$ chalice deploy
Creating deployment package.
Updating policy for IAM role: media-query-dev-detect_labels_on_image
Updating lambda function: media-query-dev-detect_labels_on_image
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪detect_labels_on_image
```

Verification

1. Run `chalice invoke` with the `sample-event.json` on the updated `detect_labels_on_image` Lambda function:

```
$ chalice invoke --name detect_labels_on_image < sample-event.json
null
```

2. Use the `get-item` CLI command to ensure the `sample.jpg` data was populated in the DynamoDB table:

```
$ aws dynamodb get-item --table-name $MEDIA_TABLE_NAME \
    --key '{"name": {"S": "sample.jpg"}}'
{
    "Item": {
        "name": {
            "S": "sample.jpg"
        },
        "labels": {
            "L": [
                {
                    "S": "Animal"
                },
                {
```

(continues on next page)

(continued from previous page)

```
                    "S": "Canine"
                },
                {
                    "S": "Dog"
                },
                {
                    "S": "German Shepherd"
                },
                {
                    "S": "Mammal"
                },
                {
                    "S": "Pet"
                },
                {
                    "S": "Collie"
                }
            ]
        },
        "type": {
            "S": "image"
        }
    }
}
```

3.5 Part 4: Add S3 event source

So far, we have been manually invoking the Lambda function ourselves in order to detect objects in the image and add the information to our database. However, we can automate this workflow using Lambda event sources so that the Lambda function is invoked every time an object is uploaded to the S3 bucket.

For this section, we will be doing the following:

- *Add Lambda event source for S3 object creation event*
- *Redeploy the Chalice application*

3.5.1 Add Lambda event source for S3 object creation event

Change the Lambda function to be invoked whenever an object is uploaded to a S3 bucket via the on_s3_event decorator.

Instructions

1. In the `app.py` file, change the `detect_labels_on_image` signature to be named `handle_object_created` that accepts a single `event` parameter:

```
def handle_object_created(event):
```

2. Update the decorator on `handle_object_created` to use the `app.on_s3_event` decorator instead and have the Lambda function be triggered whenever an object is created in the bucket specified by the environment variable `MEDIA_BUCKET_NAME`:

```
@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectCreated:*'])
def handle_object_created(event):
```

3. Add the tuple `_SUPPORTED_IMAGE_EXTENSTIONS` representing a list of supported image extensions:

```
_SUPPORTED_IMAGE_EXTENSIONS = (
    '.jpg',
    '.png',
)
```

4. Update the `handle_object_created` function to use the new `event` argument of type S3Event and only do object detection and database additions on specific image file extensions:

```
@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectCreated:*'])
def handle_object_created(event):
    if _is_image(event.key):
        _handle_created_image(bucket=event.bucket, key=event.key)


def _is_image(key):
    return key.endswith(_SUPPORTED_IMAGE_EXTENSIONS)


def _handle_created_image(bucket, key):
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)
```

Validation

1. Ensure the contents of the `app.py` file is:

```
import os

import boto3
from chalice import Chalice
from chalicelib import db
```

(continues on next page)

(continued from previous page)

```
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_MEDIA_DB = None
_REKOGNITION_CLIENT = None
_SUPPORTED_IMAGE_EXTENSIONS = (
    '.jpg',
    '.png',
)


def get_media_db():
    global _MEDIA_DB
    if _MEDIA_DB is None:
        _MEDIA_DB = db.DynamoMediaDB(
            boto3.resource('dynamodb').Table(
                os.environ['MEDIA_TABLE_NAME']))
    return _MEDIA_DB


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectCreated:*'])
def handle_object_created(event):
    if _is_image(event.key):
        _handle_created_image(bucket=event.bucket, key=event.key)


def _is_image(key):
    return key.endswith(_SUPPORTED_IMAGE_EXTENSIONS)


def _handle_created_image(bucket, key):
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)
```

3.5.2 Redeploy the Chalice application

Deploy the updated Chalice application.

Instructions

1. Run `chalice deploy`:

```
$ chalice deploy
Creating deployment package.
Creating IAM role: media-query-dev-handle_object_created
Creating lambda function: media-query-dev-handle_object_created
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_created
Deleting function: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪detect_labels_on_image
Deleting IAM role: media-query-dev-detect_labels_on_image
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_created
```

Validation

1. Upload the `othersample.jpg` image to the S3 bucket:

```
$ aws s3 cp ../chalice-workshop/code/media-query/final/assets/othersample.jpg s3:/
↪/$MEDIA_BUCKET_NAME
```

2. Use the `get-item` CLI command to ensure the `othersample.jpg` data was automatically populated in the DynamoDB table:

```
$ aws dynamodb get-item --table-name $MEDIA_TABLE_NAME \
    --key '{"name": {"S": "othersample.jpg"}}'
{
    "Item": {
        "name": {
            "S": "othersample.jpg"
        },
        "labels": {
            "L": [
                {
                    "S": "Human"
                },
                {
                    "S": "People"
                },
                {
                    "S": "Person"
                },
                {
                    "S": "Phone Booth"
                },
                {
                    "S": "Bus"
                },
```

(continues on next page)

(continued from previous page)

```
                {
                    "S": "Transportation"
                },
                {
                    "S": "Vehicle"
                },
                {
                    "S": "Man"
                },
                {
                    "S": "Face"
                },
                {
                    "S": "Leisure Activities"
                },
                {
                    "S": "Tourist"
                },
                {
                    "S": "Portrait"
                },
                {
                    "S": "Crowd"
                }
            ]
        },
        "type": {
            "S": "image"
        }
    }
}
```

If the item does not appear, try running the `get-item` command after waiting for ten seconds. Sometimes, it takes a little bit of time for the Lambda function to get triggered.

3.6 Part 5: Add S3 delete event handler

Now that we are automatically importing uploaded images to our table, we need to be able to automatically delete images from our table that get deleted from our bucket. This can be accomplished by doing the following:

- *Add Lambda function for S3 object deletion*
- *Redeploy the Chalice application*

3.6.1 Add Lambda function for S3 object deletion

Add a new Lambda function that is invoked whenever an object is deleted from the S3 bucket and if it is an image, removes the image from the table.

Instructions

1. In the `app.py` file add a new function `handle_object_removed` that is triggered whenever an object gets deleted from the bucket and deletes the item from table if it is an image:

```
@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectRemoved:*'])
def handle_object_removed(event):
    if _is_image(event.key):
        get_media_db().delete_media_file(event.key)
```

Verification

1. Ensure the contents of the `app.py` file is:

```
import os

import boto3
from chalice import Chalice
from chalicelib import db
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_MEDIA_DB = None
_REKOGNITION_CLIENT = None
_SUPPORTED_IMAGE_EXTENSIONS = (
    '.jpg',
    '.png',
)


def get_media_db():
    global _MEDIA_DB
    if _MEDIA_DB is None:
        _MEDIA_DB = db.DynamoMediaDB(
            boto3.resource('dynamodb').Table(
                os.environ['MEDIA_TABLE_NAME']))
    return _MEDIA_DB


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
```

(continues on next page)

(continued from previous page)

```
                  events=['s3:ObjectCreated:*'])
def handle_object_created(event):
    if _is_image(event.key):
        _handle_created_image(bucket=event.bucket, key=event.key)


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                  events=['s3:ObjectRemoved:*'])
def handle_object_removed(event):
    if _is_image(event.key):
        get_media_db().delete_media_file(event.key)


def _is_image(key):
    return key.endswith(_SUPPORTED_IMAGE_EXTENSIONS)


def _handle_created_image(bucket, key):
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)
```

3.6.2 Redeploy the Chalice application

Deploy the updated Chalice application with the new Lambda function.

Instructions

1. Run `chalice deploy`:

```
$ chalice deploy
Creating IAM role: media-query-dev-handle_object_removed
Creating lambda function: media-query-dev-handle_object_removed
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_removed
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_created
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_removed
```

Verification

1. Delete the uploaded `othersample.jpg` object from the previous part:

```
$ aws s3 rm s3://$MEDIA_BUCKET_NAME/othersample.jpg
```

2. Use the `scan` CLI command to ensure the object is no longer in the table:

```
$ aws dynamodb scan --table-name $MEDIA_TABLE_NAME
{
    "Items": [
        {
```

(continues on next page)

(continued from previous page)


```
            "name": {
                "S": "sample.jpg"
            },
            "labels": {
                "L": [
                    {
                        "S": "Animal"
                    },
                    {
                        "S": "Canine"
                    },
                    {
                        "S": "Dog"
                    },
                    {
                        "S": "German Shepherd"
                    },
                    {
                        "S": "Mammal"
                    },
                    {
                        "S": "Pet"
                    },
                    {
                        "S": "Collie"
                    }
                ]
            },
            "type": {
                "S": "image"
            }
        }
    ],
    "Count": 1,
    "ScannedCount": 1,
    "ConsumedCapacity": null
}
```


If the item still appears, try running the `scan` command after waiting for ten seconds. Sometimes, it takes a little bit of time for the Lambda function to get triggered. In the end, the table should only have the `sample.jpg` item.

3.7 Part 6: Add REST API to query media files

So far we have been querying the image files stored in our table via the AWS CLI. However, it would be more helpful to have an API on-top of the table instead of having to query it directly with the AWS CLI. We will now use Amazon API Gateway integrations with Lambda to create an API for our application. This API will have two routes:

- `GET /` - List all media items in the table. You can supply the query string parameters: `startswith`, `media-type`, and `label` to further filter the media items returned in the API call
- `GET /{name}` - Retrieve the media item based on the `name` of the media item.

To create this API, we will perform the following steps:

- *Add route for listing media items*
- *Add route for retrieving a single media item*
- *Redeploy the Chalice application*

3.7.1 Add route for listing media items

Add an API route `GET /` that lists all items in the table and allows users to query on `startswith`, `media-type`, and `label`.

Instructions

1. In the `app.py` file, define the function `list_media_files()` that has the route `GET /` using the app.route decorator:

```
@app.route('/')
def list_media_files():
```

2. Inside of the `list_media_files()` function, extract the query string parameters from the app.current_request object and query the database for the media files:

```
@app.route('/')
def list_media_files():
    params = {}
    if app.current_request.query_params:
        params = _extract_db_list_params(app.current_request.query_params)
    return get_media_db().list_media_files(**params)


def _extract_db_list_params(query_params):
    valid_query_params = [
        'startswith',
        'media-type',
        'label'
    ]
    return {
        k.replace('-', '_'): v
        for k, v in query_params.items() if k in valid_query_params
    }
```

Verification

1. Ensure the contents of the `app.py` file is:

```
import os

import boto3
from chalice import Chalice
from chalicelib import db
from chalicelib import rekognition

app = Chalice(app_name='media-query')
```

(continues on next page)

(continued from previous page)

```

_MEDIA_DB = None
_REKOGNITION_CLIENT = None
_SUPPORTED_IMAGE_EXTENSIONS = (
    '.jpg',
    '.png',
)


def get_media_db():
    global _MEDIA_DB
    if _MEDIA_DB is None:
        _MEDIA_DB = db.DynamoMediaDB(
            boto3.resource('dynamodb').Table(
                os.environ['MEDIA_TABLE_NAME']))
    return _MEDIA_DB


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectCreated:*'])
def handle_object_created(event):
    if _is_image(event.key):
        _handle_created_image(bucket=event.bucket, key=event.key)


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectRemoved:*'])
def handle_object_removed(event):
    if _is_image(event.key):
        get_media_db().delete_media_file(event.key)


@app.route('/')
def list_media_files():
    params = {}
    if app.current_request.query_params:
        params = _extract_db_list_params(app.current_request.query_params)
    return get_media_db().list_media_files(**params)


def _extract_db_list_params(query_params):
    valid_query_params = [
        'startswith',
        'media-type',
        'label'
    ]
    return {
        k.replace('-', '_'): v
        for k, v in query_params.items() if k in valid_query_params
```

(continues on next page)

(continued from previous page)

```
    }


def _is_image(key):
    return key.endswith(_SUPPORTED_IMAGE_EXTENSIONS)


def _handle_created_image(bucket, key):
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)
```

2. Install HTTPie to query the API:

```
$ pip install httpie
```

3. In a different terminal, run `chalice local` to run the API as a server locally:

```
$ chalice local
```

4. Use HTTPie to query the API for all images:

```
$ http 127.0.0.1:8000/
HTTP/1.1 200 OK
Content-Length: 126
Content-Type: application/json
Date: Tue, 17 Jul 2018 13:59:35 GMT
Server: BaseHTTP/0.6 Python/3.6.1

[
    {
        "labels": [
            "Animal",
            "Canine",
            "Dog",
            "German Shepherd",
            "Mammal",
            "Pet",
            "Collie"
        ],
        "name": "sample.jpg",
        "type": "image"
    }
]
```

5. Use HTTPie to query the API using the query string parameter `label`:

```
$ http 127.0.0.1:8000/ label==Dog
HTTP/1.1 200 OK
Content-Length: 126
Content-Type: application/json
Date: Tue, 17 Jul 2018 14:01:22 GMT
Server: BaseHTTP/0.6 Python/3.6.1

[
    {
        "labels": [
```

(continues on next page)

(continued from previous page)

```
            "Animal",
            "Canine",
            "Dog",
            "German Shepherd",
            "Mammal",
            "Pet",
            "Collie"
        ],
        "name": "sample.jpg",
        "type": "image"
    }
]
$ http 127.0.0.1:8000/ label==Person
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json
Date: Tue, 17 Jul 2018 14:01:46 GMT
Server: BaseHTTP/0.6 Python/3.6.1

[]
```

Feel free to test out any of the other query string parameters as well.

3.7.2 Add route for retrieving a single media item

Add an API route `GET /{name}` that retrieves a single item in the table using the `name` of the item.

Instructions

1. Import `chalice.NotFoundError` in the `app.py` file:

```
import os

import boto3
from chalice import Chalice
from chalice import NotFoundError
from chalicelib import db
from chalicelib import rekognition
```

2. In the `app.py` file, define the function `get_media_file()` decorated by `app.route('/{name}')`:

```
@app.route('/{name}')
def get_media_file(name):
```

3. Within the `get_media_file()` function, query the media item using the `name` parameter and raise a `chalice.NotFoundError` exception when the `name` does not exist in the database:

```
@app.route('/{name}')
def get_media_file(name):
    item = get_media_db().get_media_file(name)
    if item is None:
        raise NotFoundError('Media file (%s) not found' % name)
    return item
```

Verification

1. Ensure the contents of the `app.py` file is:

```
import os

import boto3
from chalice import Chalice
from chalice import NotFoundError
from chalicelib import db
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_MEDIA_DB = None
_REKOGNITION_CLIENT = None
_SUPPORTED_IMAGE_EXTENSIONS = (
    '.jpg',
    '.png',
)


def get_media_db():
    global _MEDIA_DB
    if _MEDIA_DB is None:
        _MEDIA_DB = db.DynamoMediaDB(
            boto3.resource('dynamodb').Table(
                os.environ['MEDIA_TABLE_NAME']))
    return _MEDIA_DB


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectCreated:*'])
def handle_object_created(event):
    if _is_image(event.key):
        _handle_created_image(bucket=event.bucket, key=event.key)


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectRemoved:*'])
def handle_object_removed(event):
    if _is_image(event.key):
        get_media_db().delete_media_file(event.key)


@app.route('/')
def list_media_files():
    params = {}
    if app.current_request.query_params:
        params = _extract_db_list_params(app.current_request.query_params)
```

(continues on next page)

(continued from previous page)

```
    return get_media_db().list_media_files(**params)


@app.route('/{name}')
def get_media_file(name):
    item = get_media_db().get_media_file(name)
    if item is None:
        raise NotFoundError('Media file (%s) not found' % name)
    return item


def _extract_db_list_params(query_params):
    valid_query_params = [
        'startswith',
        'media-type',
        'label'
    ]
    return {
        k.replace('-', '_'): v
        for k, v in query_params.items() if k in valid_query_params
    }


def _is_image(key):
    return key.endswith(_SUPPORTED_IMAGE_EXTENSIONS)


def _handle_created_image(bucket, key):
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)
```

2. If the local server is not still running, run `chalice local` to restart the local API server:

```
$ chalice local
```

3. Use HTTPie to query the API for the `sample.jpg` image:

```
$ http 127.0.0.1:8000/sample.jpg
HTTP/1.1 200 OK
Content-Length: 124
Content-Type: application/json
Date: Tue, 17 Jul 2018 14:09:01 GMT
Server: BaseHTTP/0.6 Python/3.6.1

{
    "labels": [
        "Animal",
        "Canine",
        "Dog",
        "German Shepherd",
        "Mammal",
        "Pet",
        "Collie"
    ],
    "name": "sample.jpg",
    "type": "image"
}
```

4. Use HTTPie to query the API for an image that does not exist:

```
$ http 127.0.0.1:8000/noexists.jpg
HTTP/1.1 404 Not Found
Content-Length: 90
Content-Type: application/json
Date: Tue, 17 Jul 2018 14:09:34 GMT
Server: BaseHTTP/0.6 Python/3.6.1

{
    "Code": "NotFoundError",
    "Message": "NotFoundError: Media file (noexists.jpg) not found"
}
```

3.7.3 Redeploy the Chalice application

Deploy the Chalice application based on the updates.

Instructions

1. Run `chalice deploy`:

```
$ chalice deploy
Creating deployment package.
Updating policy for IAM role: media-query-dev-handle_object_created
Updating lambda function: media-query-dev-handle_object_created
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_created
Updating policy for IAM role: media-query-dev-handle_object_removed
Updating lambda function: media-query-dev-handle_object_removed
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_removed
Creating IAM role: media-query-dev-api_handler
Creating lambda function: media-query-dev
Creating Rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_created
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_removed
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev
  - Rest API URL: https://1lmxgj9bfl.execute-api.us-west-2.amazonaws.com/api/
```

Verification

1. Reupload the `othersample.jpg` image using the CLI:

```
$ aws s3 cp ../chalice-workshop/code/media-query/final/assets/othersample.jpg s3:/
↪/$MEDIA_BUCKET_NAME
```

2. Use HTTPie to query the deployed API for all media items:

```
$ http $(chalice url)
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 126
Content-Type: application/json
Date: Tue, 17 Jul 2018 14:14:27 GMT
Via: 1.1 a3c7cc30af6c8465e695a3c0d44793e0.cloudfront.net (CloudFront)
X-Amz-Cf-Id: PAkgH2j5G2er_TZwyQOcwGahwNTR8dhEhrCUklcdDuuEBcKOYQ1-Ug==
X-Amzn-Trace-Id: Root=1-5b4df9c1-89a47758a7a7989e47799a12;Sampled=0
X-Cache: Miss from cloudfront
x-amz-apigw-id: KLP2SFnTPHcFeqw=
x-amzn-RequestId: b5e7488a-89cb-11e8-acbf-eda14961f501

[
    {
        "labels": [
            "Human",
            "People",
            "Person",
            "Phone Booth",
            "Bus",
            "Transportation",
            "Vehicle",
            "Man",
            "Face",
            "Leisure Activities",
            "Tourist",
            "Portrait",
            "Crowd"
        ],
        "name": "othersample.jpg",
        "type": "image"
    },
    {
        "labels": [
            "Animal",
            "Canine",
            "Dog",
            "German Shepherd",
            "Mammal",
            "Pet",
            "Collie"
        ],
        "name": "sample.jpg",
        "type": "image"
    }
]
```

Note `chalice url` just returns the URL of the remotely deployed API.

3. Use HTTPie to test out a couple of the query string parameters:

```
$ http $(chalice url) label=='Phone Booth'
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 207
Content-Type: application/json
Date: Sun, 22 Jul 2018 07:49:37 GMT
```

(continues on next page)

(continued from previous page)

```
Via: 1.1 75fd15ce5d9f38e4c444039a1548df96.cloudfront.net (CloudFront)
X-Amz-Cf-Id: nYpeS8kk_lFklCA7wCkOI0NO1wabDI3jvs3UpHFlsJ-c0nvlXNrvJQ==
X-Amzn-Trace-Id: Root=1-5b543710-8beb4000395cd60e5688841a;Sampled=0
X-Cache: Miss from cloudfront
x-amz-apigw-id: Ka2KpF0nvHcF1hg=
x-amzn-RequestId: c7e9cabf-8d83-11e8-b109-5f2c96dac9da

[
    {
        "labels": [
            "Human",
            "People",
            "Person",
            "Phone Booth",
            "Bus",
            "Transportation",
            "Vehicle",
            "Man",
            "Face",
            "Leisure Activities",
            "Tourist",
            "Portrait",
            "Crowd"
        ],
        "name": "othersample.jpg",
        "type": "image"
    }
]

$ http $(chalice url) startswith==sample
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 126
Content-Type: application/json
Date: Sun, 22 Jul 2018 07:51:03 GMT
Via: 1.1 53657f22d99084ad547a21392858391b.cloudfront.net (CloudFront)
X-Amz-Cf-Id: TORlA6wdOff5n4xHUH9ftnXNxFrTmQsSFG18acx7iwKLA_NsUoUoCg==
X-Amzn-Trace-Id: Root=1-5b543766-912f6e067cb58ddcb6a973de;Sampled=0
X-Cache: Miss from cloudfront
x-amz-apigw-id: Ka2YEGNvPHcF8SA=
x-amzn-RequestId: fb25c9e7-8d83-11e8-898d-8da83b49132b

[
    {
        "labels": [
            "Animal",
            "Canine",
            "Dog",
            "German Shepherd",
            "Mammal",
            "Pet",
            "Collie"
        ],
        "name": "sample.jpg",
        "type": "image"
    }
]
```

4. Use HTTPie to query the deployed API for `sample.jpg` image:

```
$ http $(chalice url)sample.jpg
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 124
Content-Type: application/json
Date: Tue, 17 Jul 2018 14:16:04 GMT
Via: 1.1 7ca583dd6abc0b0f42b148142a75588a.cloudfront.net (CloudFront)
X-Amz-Cf-Id: pzkZ0uZvk5e5W-ZV39v2zCCFAmmRJjDMJZ_I9GyDKhg6WEHotrMmnQ==
X-Amzn-Trace-Id: Root=1-5b4dfa24-69d586d8e94fb75019b42f24;Sampled=0
X-Cache: Miss from cloudfront
x-amz-apigw-id: KLQFrF3svHcF32Q=
x-amzn-RequestId: f0a6a6af-89cb-11e8-8420-e7ec8398ed6b

{
    "labels": [
        "Animal",
        "Canine",
        "Dog",
        "German Shepherd",
        "Mammal",
        "Pet",
        "Collie"
    ],
    "name": "sample.jpg",
    "type": "image"
}
```

3.8 Part 7: Add workflow to process videos

In the final part of this tutorial, we will add the ability to automatically process videos uploaded to our S3 bucket and add them to our DynamoDB table.

To accomplish this we will be performing the following steps:

- *Introduction to Rekognition object detection in videos*
- *Create SNS topic and IAM role*
- *Deploy a lambda function for retrieving processed video labels*
- *Automate video workflow on S3 uploads and deletions*
- *Final Code*

3.8.1 Introduction to Rekognition object detection in videos

Detecting labels in a video is a different workflow than detecting image labels when using Rekognition. Specifically, the workflow is asynchronous where you must initiate a label detection job using the StartLabelDetection API and then call GetLabelDetection once the job is complete to retrieve all of the detected labels. This step will introduce you to this workflow.

Instructions

1. Upload a sample video to the S3 bucket:

```
$ aws s3 cp ../chalice-workshop/code/media-query/final/assets/sample.mp4 s3://
↪$MEDIA_BUCKET_NAME
```

2. Run the `start-label-detection` command with the AWS CLI to start a label detection job on the uploaded video and save the `JobId`:

```
$ JOB_ID=$(aws rekognition start-label-detection --video S3Object="{Bucket=$MEDIA_
↪BUCKET_NAME,Name=sample.mp4}" --query JobId --output text)
```

3. Run the `get-label-detection` command until the `JobStatus` field is equal to `SUCCEEDED` and retrieve the video labels:

```
$ aws rekognition get-label-detection --job-id $JOB_ID
```

Verification

1. Once the `JobStatus` field is equal to `SUCCEEDED`, the output of the `get-label-detection` command should contain:


```
{
    "JobStatus": "SUCCEEDED",
    "VideoMetadata": {
        "Codec": "h264",
        "DurationMillis": 10099,
        "Format": "QuickTime / MOV",
        "FrameRate": 29.707088470458984,
        "FrameHeight": 960,
        "FrameWidth": 540
    },
    "Labels": [
        {
            "Timestamp": 0,
            "Label": {
                "Name": "Animal",
                "Confidence": 66.68909454345703
            }
        },
        {
            "Timestamp": 0,
            "Label": {
                "Name": "Dog",
                "Confidence": 60.80849838256836
            }
```


(continues on next page)

(continued from previous page)

```
        },
        {
            "Timestamp": 0,
            "Label": {
                "Name": "Husky",
                "Confidence": 51.586997985839844
            }
        },
        {
            "Timestamp": 168,
            "Label": {
                "Name": "Animal",
                "Confidence": 58.79970169067383
            }
        },
    ...[SHORTENED]...
}
```

3.8.2 Create SNS topic and IAM role

Rekognition `StartDetectLabels` also has the option to publish a message to an SNS topic once the job has completed. This is a much more efficient solution than constantly polling the `GetLabelDetection` API to wait for the labels to be detected. In this step, we will create an IAM role and SNS topic that Rekognition can use to publish this message.

Instructions

1. Copy the updated version of the `resources.json` CloudFormation template containing an IAM role and SNS topic for Rekognition to publish to:

```
$ cp ../chalice-workshop/code/media-query/07-videos/resources.json .
```

2. Deploy the new resources to your CloudFormation stack using the AWS CLI:

```
$ aws cloudformation deploy --template-file resources.json \
    --stack-name media-query --capabilities CAPABILITY_IAM
```

3. Save the SNS topic and IAM role information as environment variables in the Chalice application by running the `recordresources.py` script:

```
$ python recordresources.py --stack-name media-query
```

Verification

1. Ensure the contents of the `config.json` contains the environment variables `VIDEO_TOPIC_NAME`, `VIDEO_ROLE_ARN`, and `VIDEO_TOPIC_ARN`:

```
$ cat .chalice/config.json
{
  "version": "2.0",
  "app_name": "media-query",
  "stages": {
```

(continues on next page)

(continued from previous page)

```
    "dev": {
      "api_gateway_stage": "api",
      "autogen_policy": false,
      "environment_variables": {
        "MEDIA_TABLE_NAME": "media-query-MediaTable-10QEPR0O8DOT4",
        "MEDIA_BUCKET_NAME": "media-query-mediabucket-fb8oddjbslv1",
        "VIDEO_TOPIC_NAME": "media-query-VideoTopic-KU38EEHIIUV1",
        "VIDEO_ROLE_ARN": "arn:aws:iam::123456789123:role/media-query-VideoRole-
↪1GKK0CA30VCAD",
        "VIDEO_TOPIC_ARN": "arn:aws:sns:us-west-2:123456789123:media-query-
↪VideoTopic-KU38EEHIIUV1"
      }
    }
  }
}
```

3.8.3 Deploy a lambda function for retrieving processed video labels

With the new SNS topic, add a new Lambda function that is triggered on SNS messages to that topic, calls the `GetDetectionLabel` API, and adds the video with the labels into the database.

Instructions

1. Import `json` at the top of the `app.py` file:

```
import json
```

2. Then, define the function `add_video_file()` that uses the app.on_sns_message decorator:

```
@app.on_sns_message(topic=os.environ['VIDEO_TOPIC_NAME'])
def add_video_file(event):
```

3. Update the `add_video_file()` function, to process the `event` argument of type SNSEvent by retrieving the job ID from the message, retrieve the processed labels from Rekognition, and add the video to the database:

```
@app.on_sns_message(topic=os.environ['VIDEO_TOPIC_NAME'])
def add_video_file(event):
    message = json.loads(event.message)
    labels = get_rekognition_client().get_video_job_labels(message['JobId'])
    get_media_db().add_media_file(
        name=message['Video']['S3ObjectName'],
        media_type=db.VIDEO_TYPE,
        labels=labels)
```

3. Run `chalice deploy` to deploy the new Lambda function:

```
$ chalice deploy
Creating deployment package.
Updating policy for IAM role: media-query-dev-handle_object_created
Updating lambda function: media-query-dev-handle_object_created
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_created
Updating policy for IAM role: media-query-dev-handle_object_removed
```

(continues on next page)

(continued from previous page)

```
Updating lambda function: media-query-dev-handle_object_removed
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_removed
Creating IAM role: media-query-dev-add_video_file
Creating lambda function: media-query-dev-add_video_file
Subscribing media-query-dev-add_video_file to SNS topic media-query-VideoTopic-
↪KU38EEHIIUV1
Updating policy for IAM role: media-query-dev-api_handler
Updating lambda function: media-query-dev
Updating rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_created
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_removed
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪add_video_file
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev
  - Rest API URL: https://1lmxgj9bfl.execute-api.us-west-2.amazonaws.com/api/
```

Verification

1. Retrieve the arn of the deployed SNS topic:

```
$ VIDEO_TOPIC_ARN=$(aws cloudformation describe-stacks --stack-name media-query --
↪query "Stacks[0].Outputs[?OutputKey=='VideoTopicArn'].OutputValue" --output␣
↪text)
```

2. Retrieve the arn of the deployed IAM role:

```
$ VIDEO_ROLE_ARN=$(aws cloudformation describe-stacks --stack-name media-query --
↪query "Stacks[0].Outputs[?OutputKey=='VideoRoleArn'].OutputValue" --output text)
```

3. Run the `start-label-detection` command with the AWS CLI to start a label detection job on the uploaded video:

```
$ aws rekognition start-label-detection \
    --video S3Object="{Bucket=$MEDIA_BUCKET_NAME,Name=sample.mp4}" \
    --notification-channel SNSTopicArn=$VIDEO_TOPIC_ARN,RoleArn=$VIDEO_ROLE_ARN
```

4. Wait roughly twenty seconds and then use HTTPie to query for the video against the application's API:

```
$ http $(chalice url)sample.mp4
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 151
Content-Type: application/json
Date: Tue, 17 Jul 2018 21:42:12 GMT
Via: 1.1 aa42484f82c16d99015c599631def20c.cloudfront.net (CloudFront)
X-Amz-Cf-Id: GpqmQOwnKcaxb2sP2fi-KSs8LCu24Q6ekKV8Oyo6a0HZ7kcnSGMpnQ==
X-Amzn-Trace-Id: Root=1-5b4e62b4-da9db3b1e4c95470cbc2b160;Sampled=0
X-Cache: Miss from cloudfront
x-amz-apigw-id: KMRcNHUQvHcFaDQ=
x-amzn-RequestId: 43c1cb91-8a0a-11e8-af84-8901f225e7d3
```

(continues on next page)

(continued from previous page)

```
{
    "labels": [
        "Clothing",
        "Bird Nest",
        "Dog",
        "Human",
        "People",
        "Person",
        "Husky",
        "Animal",
        "Nest",
        "Footwear"
    ],
    "name": "sample.mp4",
    "type": "video"
}
```

5. Make sure the `sample.mp4` is included when querying for items that have a `video` media type:

```
$ http $(chalice url) media-type==video
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 153
Content-Type: application/json
Date: Sun, 22 Jul 2018 07:58:28 GMT
Via: 1.1 5d53b9570a535c2d94ce93c20abbd471.cloudfront.net (CloudFront)
X-Amz-Cf-Id: JwvyQ_rEePlEyRAGjtQ1jDnvjXPKt8ea3FiNLdgBbjWnf2G4UTpUaw==
X-Amzn-Trace-Id: Root=1-5b543923-02ddf1e74491eb77d692c8fd;Sampled=0
X-Cache: Miss from cloudfront
x-amz-apigw-id: Ka3dkFlHvHcFYIQ=
x-amzn-RequestId: 0441fc0a-8d85-11e8-b51a-bd624fe1291d

[
    {
        "labels": [
            "Footwear",
            "Human",
            "People",
            "Nest",
            "Bird Nest",
            "Person",
            "Dog",
            "Husky",
            "Clothing",
            "Animal"
        ],
        "name": "sample.mp4",
        "type": "video"
    }
]
```

3.8.4 Automate video workflow on S3 uploads and deletions

Now let's update the application so we do not have to manually invoke the `StartLabelDetection` API and instead have the API be invoked in Lambda whenever a video is uploaded to S3. We will also need to automatically delete the video whenever the video is deleted from S3.

Instructions

1. Add the tuple `_SUPPORTED_VIDEO_EXTENSTIONS` representing a list of supported video extensions:

```
_SUPPORTED_VIDEO_EXTENSIONS = (
    '.mp4',
    '.flv',
    '.mov',
)
```

2. Update the `handle_object_created` function to start a video label detection job for videos uploaded to the S3 bucket and have the completion notification be published to the SNS topic:

```
@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectCreated:*'])
def handle_object_created(event):
    if _is_image(event.key):
        _handle_created_image(bucket=event.bucket, key=event.key)
    elif _is_video(event.key):
        _handle_created_video(bucket=event.bucket, key=event.key)


def _is_video(key):
    return key.endswith(_SUPPORTED_VIDEO_EXTENSIONS)


def _handle_created_video(bucket, key):
    get_rekognition_client().start_video_label_job(
        bucket=bucket, key=key, topic_arn=os.environ['VIDEO_TOPIC_ARN'],
        role_arn=os.environ['VIDEO_ROLE_ARN']
    )
```

3. Update the `handle_object_removed` function to delete items from the table that are videos as well:

```
@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectRemoved:*'])
def handle_object_removed(event):
    if _is_image(event.key) or _is_video(event.key):
        get_media_db().delete_media_file(event.key)
```

4. Run `chalice deploy` to deploy the updated Chalice application:

```
$ chalice deploy
Creating deployment package.
Updating policy for IAM role: media-query-dev-handle_object_created
Updating lambda function: media-query-dev-handle_object_created
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_created
Updating policy for IAM role: media-query-dev-handle_object_removed
Updating lambda function: media-query-dev-handle_object_removed
```

(continues on next page)

(continued from previous page)

```
Configuring S3 events in bucket media-query-mediabucket-fb8oddjbslv1 to function␣
↪media-query-dev-handle_object_removed
Creating IAM role: media-query-dev-add_video_file
Creating lambda function: media-query-dev-add_video_file
Subscribing media-query-dev-add_video_file to SNS topic media-query-VideoTopic-
↪KU38EEHIIUV1
Updating policy for IAM role: media-query-dev-api_handler
Updating lambda function: media-query-dev
Updating rest API
Resources deployed:
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_created
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_removed
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪add_video_file
  - Lambda ARN: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev
  - Rest API URL: https://1lmxgj9bfl.execute-api.us-west-2.amazonaws.com/api/
```

Verification

1. Delete the previously uploaded `sample.mp4` from the S3 bucket:

```
$ aws s3 rm s3://$MEDIA_BUCKET_NAME/sample.mp4
```

2. Ensure the `sample.mp4` video no longer is queryable from the application's API:

```
$ http $(chalice url)sample.mp4
HTTP/1.1 404 Not Found
Connection: keep-alive
Content-Length: 88
Content-Type: application/json
Date: Tue, 17 Jul 2018 22:06:57 GMT
Via: 1.1 e93b65cf89966087a2d9723b4713fb37.cloudfront.net (CloudFront)
X-Amz-Cf-Id: XD7Wr8-zY8cUAEvnSU_ojyvAadTiNatcJXuztSmBta3Kiluvuvf6ug==
X-Amzn-Trace-Id: Root=1-5b4e6880-c6c366d38f1e906798146b4b;Sampled=0
X-Cache: Error from cloudfront
x-amz-apigw-id: KMVEAFEPPHcFieQ=
x-amzn-RequestId: b7fba401-8a0d-11e8-a7e4-a9e75b4bb382

{
    "Code": "NotFoundError",
    "Message": "NotFoundError: Media file (sample.mp4) not found"
}
```

3. Reupload the `sample.mp4` to the S3 bucket:

```
$ aws s3 cp ../chalice-workshop/code/media-query/final/assets/sample.mp4 s3://
↪$MEDIA_BUCKET_NAME
```

4. After waiting roughly 20 seconds, ensure the `sample.mp4` video is queryable again from the application's API:

```
$ http $(chalice url)sample.mp4
HTTP/1.1 200 OK
```

(continues on next page)

(continued from previous page)

```
Connection: keep-alive
Content-Length: 151
Content-Type: application/json
Date: Tue, 17 Jul 2018 21:42:12 GMT
Via: 1.1 aa42484f82c16d99015c599631def20c.cloudfront.net (CloudFront)
X-Amz-Cf-Id: GpqmQOwnKcaxb2sP2fi-KSs8LCu24Q6ekKV8Oyo6a0HZ7kcnSGMpnQ==
X-Amzn-Trace-Id: Root=1-5b4e62b4-da9db3b1e4c95470cbc2b160;Sampled=0
X-Cache: Miss from cloudfront
x-amz-apigw-id: KMRcNHUQvHcFaDQ=
x-amzn-RequestId: 43c1cb91-8a0a-11e8-af84-8901f225e7d3

{
    "labels": [
        "Clothing",
        "Bird Nest",
        "Dog",
        "Human",
        "People",
        "Person",
        "Husky",
        "Animal",
        "Nest",
        "Footwear"
    ],
    "name": "sample.mp4",
    "type": "video"
}
```

3.8.5 Final Code

Congratulations! You have now completed this tutorial. Below is the final code that you should have wrote in the `app.py` of your Chalice application:

```
import json
import os

import boto3
from chalice import Chalice
from chalice import NotFoundError
from chalicelib import db
from chalicelib import rekognition

app = Chalice(app_name='media-query')

_MEDIA_DB = None
_REKOGNITION_CLIENT = None
_SUPPORTED_IMAGE_EXTENSIONS = (
    '.jpg',
    '.png',
)
_SUPPORTED_VIDEO_EXTENSIONS = (
    '.mp4',
    '.flv',
    '.mov',
)
```

(continues on next page)

(continued from previous page)

```
def get_media_db():
    global _MEDIA_DB
    if _MEDIA_DB is None:
        _MEDIA_DB = db.DynamoMediaDB(
            boto3.resource('dynamodb').Table(
                os.environ['MEDIA_TABLE_NAME']))
    return _MEDIA_DB


def get_rekognition_client():
    global _REKOGNITION_CLIENT
    if _REKOGNITION_CLIENT is None:
        _REKOGNITION_CLIENT = rekognition.RekognitonClient(
            boto3.client('rekognition'))
    return _REKOGNITION_CLIENT


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectCreated:*'])
def handle_object_created(event):
    if _is_image(event.key):
        _handle_created_image(bucket=event.bucket, key=event.key)
    elif _is_video(event.key):
        _handle_created_video(bucket=event.bucket, key=event.key)


@app.on_s3_event(bucket=os.environ['MEDIA_BUCKET_NAME'],
                 events=['s3:ObjectRemoved:*'])
def handle_object_removed(event):
    if _is_image(event.key) or _is_video(event.key):
        get_media_db().delete_media_file(event.key)


@app.on_sns_message(topic=os.environ['VIDEO_TOPIC_NAME'])
def add_video_file(event):
    message = json.loads(event.message)
    labels = get_rekognition_client().get_video_job_labels(message['JobId'])
    get_media_db().add_media_file(
        name=message['Video']['S3ObjectName'],
        media_type=db.VIDEO_TYPE,
        labels=labels)


@app.route('/')
def list_media_files():
    params = {}
    if app.current_request.query_params:
        params = _extract_db_list_params(app.current_request.query_params)
    return get_media_db().list_media_files(**params)


@app.route('/{name}')
def get_media_file(name):
    item = get_media_db().get_media_file(name)
    if item is None:
```

(continues on next page)

(continued from previous page)

```
        raise NotFoundError('Media file (%s) not found' % name)
    return item


def _extract_db_list_params(query_params):
    valid_query_params = [
        'startswith',
        'media-type',
        'label'
    ]
    return {
        k.replace('-', '_'): v
        for k, v in query_params.items() if k in valid_query_params
    }


def _is_image(key):
    return key.endswith(_SUPPORTED_IMAGE_EXTENSIONS)


def _handle_created_image(bucket, key):
    labels = get_rekognition_client().get_image_labels(bucket=bucket, key=key)
    get_media_db().add_media_file(key, media_type=db.IMAGE_TYPE, labels=labels)


def _is_video(key):
    return key.endswith(_SUPPORTED_VIDEO_EXTENSIONS)


def _handle_created_video(bucket, key):
    get_rekognition_client().start_video_label_job(
        bucket=bucket, key=key, topic_arn=os.environ['VIDEO_TOPIC_ARN'],
        role_arn=os.environ['VIDEO_ROLE_ARN']
    )
```

Feel free to add your own media files and/or build additional logic on top of this application. For the complete final application, see the GitHub repository

3.9 Cleaning up the Chalice application

This part of the tutorial provides instructions on how you can clean up your deployed resources once you are done using this application. This set of instructions can be completed at any point during the tutorial to clean up the application.

3.9.1 Instructions

1. Delete the chalice application:

```
$ chalice delete
Deleting Rest API: kyfn3gqcf0
Deleting function: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev
Deleting IAM role: media-query-dev-api_handler
Deleting function: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪add_video_file
```

(continues on next page)

(continued from previous page)

```
Deleting IAM role: media-query-dev-add_video_file
Deleting function: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_removed
Deleting IAM role: media-query-dev-handle_object_removed
Deleting function: arn:aws:lambda:us-west-2:123456789123:function:media-query-dev-
↪handle_object_created
Deleting IAM role: media-query-dev-handle_object_created
```

2. Delete all objects in your S3 bucket:

```
$ aws s3 rm s3://$MEDIA_BUCKET_NAME --recursive
delete: s3://media-query-mediabucket-4b1h8anboxpa/sample.jpg
delete: s3://media-query-mediabucket-4b1h8anboxpa/sample.mp4
```

3. Delete the CloudFormation stack containing the additional AWS resources:

```
$ aws cloudformation delete-stack --stack-name media-query
```

3.9.2 Validation

1. Ensure that the API for the application no longer exists:

```
$ chalice url
Error: Could not find a record of a Rest API in chalice stage: 'dev'
```

2. Check the existence of a couple of resources from the CloudFormation stack to make sure the resources no longer exist:

```
$ aws s3 ls s3://$MEDIA_BUCKET_NAME
An error occurred (NoSuchBucket) when calling the ListObjects operation: The␣
↪specified bucket does not exist

$ aws dynamodb describe-table --table-name $MEDIA_TABLE_NAME
An error occurred (ResourceNotFoundException) when calling the DescribeTable␣
↪operation: Requested resource not found: Table: media-query-MediaTable-
↪YIM7BMEIOF8Y not found
```