Creating Get Songs Service with Flask

Estimated time needed: 90 minutes

Welcome to the **Creating Get Songs Service with Flask** hands-on lab. In this lab, you will begin to build the service that you will eventually deploy to IBM Code Engine. The lab provides a GitHub template repository to get you started. The repository also contains python unit tests. You are being asked to complete the code so that the code passes all the tests.

Objectives

In this lab, you will:

- Start MongoDB database server
- Create a Flask server
- Write RESTful APIs on song resource
- Test the APIs

Note: Important Security Information

Note: Important Security Information

Welcome to the Cloud IDE. This is where all your development will take place. It has all the tools you will need to use, including **Python** and **Flask**.

It is important to understand that the lab environment is ephemeral. It only lives for a short while before it is destroyed. It is imperative that you push all changes made to your own GitHub repository so that it can be recreated in a new lab environment any time it is required.

Also, note that this environment is shared and, therefore, not secure. You should not store any personal information, usernames, passwords, or access tokens in this environment for any purpose.

Your Task

If you haven't generated a GitHub Personal Access Token you should do so now. You will need it to push code back to your repository. It should have `repo` and `write` permissions, and be set to expire in `60` days. When Git prompts you for a password in the Cloud IDE environment, use your Personal Access Token instead. Follow the steps in the Generating Git Token Lab for detailed instructions.

The environment may be recreated at any time, so you may find that you have to perform the Initialize Development Environment each time the environment is created.

Note on Screenshots

Throughout this lab, you will be prompted to take screenshots and save them on your device. You will need these screenshots to either answer graded quiz questions or upload the screenshots as your submission for peer review at the end of this course. Your screenshot must have either the .jpg or .png extension.

To take screenshots, you can use various free screen-capture tools or your operating system's shortcut keys. For example:

- Mac: you can use `Shift + Command + 3 (⇧ + ⌘ + 3)` on your keyboard to capture your entire screen, or `Shift + Command + 4 (⇧ + ⌘ + 4)` to capture a window or area. It will be saved as a .jpg or .png file on your Desktop.
- Windows: you can capture your active window by pressing `Alt + Print Screen` on your keyboard. This command copies an image of your active window to the clipboard. Next, open an image editor, paste the image from your clipboard to the image editor, and save the image as .jpg or .png.

Initialize Development Environment

Because the Cloud IDE environment is ephemeral, it may be deleted at any time. The next time you come into the lab, a new environment may be created. Unfortunately, this means that you will need to initialize your development environment every time it is recreated. This shouldn't happen too often as the environment can last for several days at a time but when it is removed, this is the procedure to recreate it.

Overview

Create new repository from template

1. Click this URL to open the starter code project: https://github.com/ibm-developer-skills-network/hvlga-Back-End-Development-Songs
2. Use the green **Use this template** button to clone this repository to your private GitHub account.

   Do not use Fork; use the Template button.

3. Give your repository the name `Back-End-Development-Songs`. This is the name that graders will look for to grade your work.
4. Ensure you select the Public option for your repository and then create it.

Initialize Development Environment

Each time you need to set up your lab development environment you will need to run three commands.

Each command will be explained in further detail, one at a time.

The commands include:

```
git clone https://github.com/$GITHUB_ACCOUNT/Back-End-Development-Songs.git
cd /home/project/Back-End-Development-Songs
bash ./bin/setup.sh
exit
```

Now, let's discuss each of these commands and explain what needs to be done.

Task Details

Initialize your environment using the following steps:

1. Open a terminal with `Terminal -> New Terminal` if one is not open already.
2. Next, use the export GITHUB_ACCOUNT command to export an environment variable that contains the name of your GitHub account.

   Note: Substitute your real GitHub account for the {your_github_account} placeholder below:

```
export GITHUB_ACCOUNT={your_github_account}
```

3. Then use the following commands to clone your repository.

```
git clone https://github.com/$GITHUB_ACCOUNT/Back-End-Development-Songs.git
```

4. Change into the devops-capstone-project directory, and execute the `./bin/setup.sh` command.

```
cd /home/project/Back-End-Development-Songs
bash ./bin/setup.sh
```

5. You should see the follow at the end of the setup execution:

6. Finally, use the `exit` command to close the current terminal. The environment will not be fully active until you open a new terminal in the next step.

```
exit
```

Validate

To validate that your environment is working correctly, you must open a new terminal because the Python virtual environment will only activate when a new terminal is created. You should have ended the previous task by using the `exit` command to exit the terminal.

1. Open a terminal with `Terminal` -> `New Terminal` and check that everything worked correctly by using the `which python` command:

   Check which Python you are using:

   ```
   which python
   ```

   You should get back:

   Check the Python version:

   ```
   python --version
   ```

   You should get back some patch level of Python 3.9.18:

Evidence

1. Note down the URL of your GitHub repository (not the template) to submit for peer review. Recall the graders are looking for a repository named `Back-End-Development-Songs` in your account.

This completes the setup of the development environment. Anytime your environment is recreated, you will need to follow this procedure.

Task 2 - Start MongoDB Server

Start the MongoDB server using the following steps:

1. Open the `Databases` tab and click `MongoDB`.

2. Click the `Create` button to start the MongoDB Database server.

3. The MongoDB server should be active now. Open the `Connection Information` tab and keep the `Password` safe. You will be using them in this lab later. Also make note of the `Host` and `Port`. You will need this information to connect to the database.

You are now ready to start working. When you start the lab environment next time, ensure the MongDB service is `ACTIVE`. If you need to restart the database, the password might change.

Project Overview

In the last module, you built the pictures service as a microservice in Flask. In this lab, you are asked to continue working on the band website. You will build a songs microservice in Flask.

This microservice works with MongoDB database to store lyrics of the most popular songs of the band. You will be using the `PyMongo` python module to interact with MongoDB programatically.

REST API Guidelines Review

The architect has provided you with the following schema for the endpoints:

RESTful API Endpoints

Action	Method	Return code	Body	URL Endpoint
List	GET	200 OK	Array of songs `[{...}]`	`GET /song`
Create	POST	201 CREATED	A song resource as json `{...}`	POST `/song`
Read	GET	200 OK	A song as json `{...}`	GET `/song/{id}`
Update	PUT	200 OK	A song as json `{...}`	PUT `/song/{id}`
Delete	DELETE	204 NO CONTENT	`""`	DELETE `/song/{id}`
Health	GET	200 OK	`""`	GET `/health`
Count	GET	200 OK	`""`	GET `/count`

Exercise 1: Write health and count endpoints

As in the pictures microservice, you need to implement the two endpoints of:

- `/health`
- `/count`

The health endpoint will simply return a JSON object with a message of `{"status": "OK"}`. The count endpoint will count the number of documents in the songs collections of the band database.

You will start the server and simply use the curl command to test all the endpoints in this lab. Open the terminal if you don't have it open already and change into the GitHub repository directory.

```
cd /home/project/Back-End-Development-Songs
```

Next, run the following command to run the flask server in development mode:

```
MONGODB_SERVICE=localhost MONGODB_USERNAME=root MONGODB_PASSWORD=password flask --app app run --debugger --reload
```

Replace `MONGODB_SERVICE` and `MONGODB_PASSWORD` with your own values. The `MONGODB_USERNAME` variable should stay as `root`.

Since your main application is in a file called `app.py`, you don't have to specify it. The following command has the same result:

```
MONGODB_SERVICE=localhost MONGODB_USERNAME=root MONGODB_PASSWORD=password flask run --reload --debugger
```

You should see the flask server running with the following output in the terminal:

```
$ (backend-songs-venv) theia:private-get-songs$ MONGODB_SERVICE=127.0.0.1 MONGODB_USERNAME=root MONGODB_PASSWORD=NDQwMC1jYXB0YWlu flask run --reload --debugger
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
The value of MONGODB_SERVICE is: 127.0.0.1
connecting to url: mongodb://root:NDQwMC1jYXB0YWlu@127.0.0.1
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
```

Task 1 : Create a Branch

Since you are working in branches, you must pull the latest changes from the main branch to stay up to date. You can then create a new branch. If you are still running the server in the first terminal, use the split button to open another terminal and execute the following:

The steps are:

```
cd /home/project/Back-End-Development-Songs
git checkout main
git pull
git checkout -b backend-rest
```

This will switch to the main branch, pull the latest changes, and create a new branch. You will be asked to push all your changes to your GitHub repo and merge all code back into your main branch with a pull request.

You can use the `git branch` command to see your current branch:

```
git branch
```

Your output should look something like this:

```
$ git branch
* backend-rest
  main
```

Task 2 : Implement the /health endpoint

1. Create the `/health` endpoint. You will write all the code in the `Back-End-Development-Songs/backend/routes.py` file.

Open **routes.py** in IDE

▼ Click here for a hint.

```
@app.route("{insert URL here}", methods="{insert HTTP method name here}")
def {insert method name here}():
    return {insert data list here}
```

Note that the flask server is running in the first terminal. Open a second terminal and execute the following command to test the endpoint:

```
curl -X GET -i -w '\n' localhost:5000/health
```

You should see the following output:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Wed, 08 Feb 2023 16:37:14 GMT
Content-Type: application/json
Content-Length: 16
Connection: close
{"status":"OK"}
```

Evidence

1. Take a screenshot of the terminal after executing the curl command. Save the screenshot as `songs-ex1-health-curl.jpg` (or `.png`).

Task 3 : Implement the /count endpoint

1. Create the `/count` endpoint in the `Back-End-Development-Songs/backend/routes.py` file.

▼ Click here for a hint.

```
@app.route("{insert url here}")
def count():
    """return length of data"""
    count = {use count_documents method here}
    return {"count": count}, {insert HTTP OKAY response code here}
```

You can test the endpoint with the following CURL command:

```
curl -X GET -i -w '\n' localhost:5000/count
```

You should see the following output:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Wed, 08 Feb 2023 16:40:08 GMT
Content-Type: application/json
Content-Length: 13
Connection: close
{"count":20}
```

Exercise 2: Implement the GET /song endpoint

You will implement the GET /song endpoint in this exercise. `PyMongo` provides `db.collections.find({})` method to get all documents in ths songs collection.

Your Task

Before you write the code for the endpoint, let's create a branch so you can commit your code back to GitHub.

Task 1 : Finish the code for the endpoint

As before, you will write the code for the endpoint in the `Back-End-Development-Songs/backend/routes.py` file.

Open **routes.py** in IDE

1. Create a Flask route that responds to the GET method for the endpoint `/song`.
2. Create a function called `songs` to hold the implementation.
3. Call the `db.songs.find({})` which will return all documents in the database.
4. Send the data as a list in the form of `{"songs":list of songs}` and a return code of HTTP_200_OK back to the caller.
5. Ensure the flask server is already running. Run the following curl command to test if the method worked:

```
curl --request GET -i -w '\n' --url http://localhost:5000/song
```

You should see an ouput similar to:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:22:09 GMT
Content-Type: application/json
Content-Length: 6694
Connection: close
{"songs":[{"_id":{"$oid":"63e4587241323a01d2058e0c"},"id":1,"lyrics":"Morbi non lectus. Aliquam sit amet diam in magna bibendum imperdiet. Nullam orci pede, venenatis non, sodales sed, tincidunt eu, felis.","t
...
...
```

Evidence

Take a screenshot of the curl command result in the terminal and name it `songs-ex2-get-song-curl.jpg` (or `.png`).

Great, you added the second endpoint to your implementation.

Exercise 3: Implement the GET /song/id endpoint

You will implement the GET /song/id endpoint in this exercise. `PyMongo` provides the `db.collections.find({})` method to get a specific document from the Mongo Database. You can look by any property to find the document. You will pass in the `id` property.

Your Task

As before, you will write the code for the endpoint in the `Back-End-Development-Songs/backend/routes.py` file.

Open **routes.py** in IDE

Note: To open in File Explorer, go to this location:
`Back-End-Development-Songs/backend/routes.py`

1. Create a Flask route that responds to the GET method for the endpoint `/song/<id>`.
2. Create a function called `get_song_by_id(id)` to hold the implementation.
3. Use the `db.songs.find_one({"id": id})` method to find a song by id
4. Return a message of `{"message": "song with id not found"}` with an HTTP code of 404 NOT FOUND if the id is not found.
5. Return the song as json with a status of 200 HTTP OK if you find the song in the database
6. Ensure the Flask server is running. Run the following curl command to test the implementation:

```
curl --request GET -i -w '\n' --url http://localhost:5000/song/1
```

You should see an ouput similar to the following:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:23:59 GMT
Content-Type: application/json
Content-Length: 256
Connection: close
{"_id":{"$oid":"63e4587241323a01d2058e0c"},"id":1,"lyrics":"Morbi non lectus. Aliquam sit amet diam in magna bibendum imperdiet. Nullam orci pede, venenatis non, sodales sed, tincidunt eu, felis.","title":"dui
```

Let's move to the next endpoint.

Exercise 4: Implement the POST /song endpoint

You will use the `db.songs.insert_one` method available in `PyMongo` to insert a single song into the database. You will first extract the song from the request body.

Your Task

As before, you will write the code for the endpoint in the `Back-End-Development-Songs/backend/routes.py` file.

Open **routes.py** in IDE

Note: To open in File Explorer, go to this location:
`Back-End-Development-Songs/backend/routes.py`

1. Create a Flask route that responds to the POST method for the endpoint `/song`. Use the `methods=["POST"]` in your app decorator.
2. Create a function called `create_song()` to hold the implementation.
3. You will first extract the song data from the request body and then append it to the `data` list.
4. If a song with the id already exists, send an HTTP code of `302` back to the user with a message of `{"Message": "song with id {song['id']} already present"}`.
5. Run the following curl command to test the implementation:

```
curl --request POST \
  -i -w '\n' \
  --url http://localhost:5000/song \
  --header 'Content-Type: application/json' \
  --data '{
        "id": 323,
        "lyrics": "Integer tincidunt ante vel ipsum. Praesent blandit lacinia erat. Vestibulum sed magna at nunc commodo placerat.\n\nPraesent blandit. Nam nulla. Integer pede justo, lacinia eget, tincidunt eg
        "title": "in faucibus orci luctus et ultrices"
    }'
```

You should see an output similar to:

```
HTTP/1.1 201 CREATED
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:26:43 GMT
Content-Type: application/json
Content-Length: 52
Connection: close
{"inserted id":{"$oid":"63e459e3b22f516761d30171"}}
```

If you send the same command again, the server should send back 302 to the curl client:

```
HTTP/1.1 302 FOUND
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:26:52 GMT
Content-Type: application/json
Content-Length: 47
Connection: close
{"Message":"song with id 323 already present"}
```

Exercise 5: Implement the PUT /song endpoint

You are asked to update a song in this endpoint. The client will send the updated song in the request body. You will use the `db.songs.update_one()` method in `PyMongo` to implement this method. Recall that the `update_one` method takes a `$set` argument as the changed song.

Your Task

The PUT endpoint will be used to update an existing picture resource. As before, you will write the code for the endpoint in the `Back-End-Development-Songs/backend/routes.py` file.

Open **routes.py** in IDE

Note: To open in File Explorer, go to this location:
`Back-End-Development-Songs/backend/routes.py`

1. Create a Flask route that responds to the POST method for the endpoint `/song/<int:id>`. Use the `methods=["PUT"]` in your app decorator.
2. Create a function called `update_song(id)` to hold the implementation.
3. You will first need to extract the song data from the request body.
4. You will then find the song in the database using the `db.songs.find_one` method. If the song exists, you will update it with the incoming request using the `db.songs.update_one` method.
5. If the song does not exist, you will send back a status of `404` with a message of `{"message": "song not found"}`.
6. Run the following curl command to test the implementation:

```
curl --request PUT \
  -i -w '\n' \
  --url http://localhost:5000/song/1 \
  --header 'Content-Type: application/json' \
  --data '{
        "lyrics": "yay hey yay yay",
        "title": "yay song"
    }'
```

You should see an output similar to:

```
HTTP/1.1 201 CREATED
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:45:21 GMT
Content-Type: application/json
Content-Length: 97
Connection: close
{"_id":{"$oid":"63e459e1b22f516761d3015d"},"id":1,"lyrics":"yay hey yay yay","title":"yay song"}
```

If you make the exact same call again, you should see the following output:

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:45:40 GMT
Content-Type: application/json
Content-Length: 46
Connection: close
{"message":"song found, but nothing updated"}
```

Evidence

Take a screenshot of the curl command result in the terminal and name it `songs-ex5--put-song-passing.jpg` (or `.png`).

Exercise 6: Implement the DELETE /song endpoint

You will impement the `DELETE song` endpoint in this exercise. You will use the `db.songs.delete_one` method provided by `PyMongo` for this purpose. You can check if any documents were modified by the method using the `deleted_count` property. If `deleted_count` is 0, the document was not found in the collection.

Task 1 : Implement the Delete endpoint

The DELETE endpoint is used to delete an existing song resource. As before, you will write the code for the endpoint in the `Back-End-Development-Songs/backend/routes.py` file.

Open **routes.py** in IDE

Note: To open in File Explorer, go to this location:
`Back-End-Development-Songs/backend/routes.py`

1. Create a Flask route that responds to the POST method for the endpoint `/song/<int:id>`. Use the `methods=["DELETE"]` in your app decorator.
2. Create a function called `delete_song(id)` to hold the implementation.
3. You will first extract the id from the URL.
4. Next use the `db.songs.delete_one` method to delete the song from the database.
5. Check the `deleted_count` attribute of the result. If the `deleted_count` is zero, you will send back a status of `404` with a message of `{"message": "song not found"}`.
6. If the `deleted_count` is 1, it means the song was succesfully deleted. You will return an empty body with a status of HTTP_204_NO_CONTENT.
7. Run the following curl command to test the implementation:

```
curl --request DELETE \
  -i -w '\n' \
  --url http://localhost:5000/song/14 \
  --header 'Content-Type: "application/json"'
```

You should see an output similar to:

```
HTTP/1.1 204 NO CONTENT
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:56:02 GMT
Content-Type: text/html; charset=utf-8
Connection: close
```

If you make the exact same call again, you should see a response of 404 as the song was already deleted by the last call:

```
HTTP/1.1 404 NOT FOUND
Server: Werkzeug/2.2.2 Python/3.7.16
Date: Thu, 09 Feb 2023 02:56:15 GMT
Content-Type: application/json
Content-Length: 29
Connection: close
{"message":"song not found"}
```

Evidence

Take a screenshot of the curl command result in the terminal and name it `songs-ex6-delete-song-passing.jpg` (or `.png`).

Task 2 : Push the branch to GitHub and create a PR

Now that you have finished the code for the microservice, you can push the `backend-rest` branch back to your GitHub fork. Since you are the only one working on this project, go ahead and merge the PR and delete the branch. Make sure all your code changes are pushed back to the main branch before proceeding to the next lab.

1. Use the `git commit -am` command to commit your changes with the message “implemented songs service”, and the `git push` command to push those changes to your repository.

   Note: You will be prompted to set up your git user and email the first time you push:

   ```
   git config --local user.name "{your GitHub name here}"
   git config --local user.email {your GitHub email here}
   ```

▼ Click here for a hint.

```
git commit -am "{message here}"
git push --set-upstream origin {branch name here}
```

▼ Click here for a hint.

```
git commit -am "implemented songs service"
git push --set-upstream origin backend-rest
```

2. Create a pull request on GitHub to merge your changes into the main branch, and, since there is no one else on your team, accept the pull request, merge it, and delete the branch.

The main branch, at this point, should have your completed code.

Solutions

This page contains the solutions for the List, Create, Update, and Delete REST APIs.

Solutions

Health

▼ Click here to check your solution.

```
@app.route("/health")
def healthz():
    return jsonify(dict(status="OK")), 200
```

Count

▼ Click here to check your solution.

```
@app.route("/count")
def count():
    """return length of data"""
    count = db.songs.count_documents({})
    return {"count": count}, 200
```

List

▼ Click here to check your solution.

```
@app.route("/song", methods=["GET"])
def songs():
    # docker run -d --name mongodb-test -e MONGO_INITDB_ROOT_USERNAME=user
    # -e MONGO_INITDB_ROOT_PASSWORD=password -e MONGO_INITDB_DATABASE=collection mongo
    results = list(db.songs.find({}))
    print(results[0])
    return {"songs": parse_json(results)}, 200
```

Read

▼ Click here to check your solution.

```
@app.route("/song/<int:id>", methods=["GET"])
def get_song_by_id(id):
    song = db.songs.find_one({"id": id})
    if not song:
        return {"message": f"song with id {id} not found"}, 404
    return parse_json(song), 200
```

Create

▼ Click here to check your solution.

```
@app.route("/song", methods=["POST"])
def create_song():
    # get data from the json body
    song_in = request.json
    print(song_in["id"])
    # if the id is already there, return 303 with the URL for the resource
    song = db.songs.find_one({"id": song_in["id"]})
    if song:
        return {
            "Message": f"song with id {song_in['id']} already present"
```

```
        }, 302
    insert_id: InsertOneResult = db.songs.insert_one(song_in)
    return {"inserted id": parse_json(insert_id.inserted_id)}, 201
```

Update

▼ Click here to check your solution.

```
@app.route("/song/<int:id>", methods=["PUT"])
def update_song(id):
    # get data from the json body
    song_in = request.json
    song = db.songs.find_one({"id": id})
    if song == None:
        return {"message": "song not found"}, 404
    updated_data = {"$set": song_in}
    result = db.songs.update_one({"id": id}, updated_data)
    if result.modified_count == 0:
        return {"message": "song found, but nothing updated"}, 200
    else:
        return parse_json(db.songs.find_one({"id": id})), 201
```

Delete

▼ Click here to check your solution.

```
@app.route("/song/<int:id>", methods=["DELETE"])
def delete_song(id):
    result = db.songs.delete_one({"id": id})
    if result.deleted_count == 0:
        return {"message": "song not found"}, 404
    else:
        return "", 204
```

Conclusion

Congratulations! You have finished implementing the second microservice for getting songs. This microservice will be used by the main site in the final lab for the project.

Next Steps

You can resume the course at this point. You will be asked to create the main Django application in the next module.

Author(s)

CF