Homework 3

AWS Lambda and API Gateway

Brian Prost

Dr. Erroll Waithe

SDEV 400 7380

30 November 2021

Part One: Data & API

Grab data

I'll be using hockey, baseball, and basketball for this. So the data will be current data, as of time of writing (morning of November 30). The data for each team's schedule history will be:

- game id (number)
- date (string)
- opponent (string)
- win or loss (boolean)
- points for (number)
- points against (number)

They will be stored in JSON files. Each looks like this:

```
[
  {
      "game_id": 1,
      "date": "Nov 27, 2021",
      "opponent": "Columbus Blue Jackets",
      "game_won": true,
      "points_for": 6,
      "points_against": 3
  },
  {
      "game_id": 2,
      "date": "Nov 26, 2021",
      "opponent": "Chicago Blackhawks",
      "game_won": false,
      "points_for": 2,
      "points_against": 3
  },
...
]
```

They will be stored in a directory called `game_data`, of which has three folders, one for each sport. Each sport folder contains three .JSON files; one for each team. These will be stored in the lambda environment on AWS.

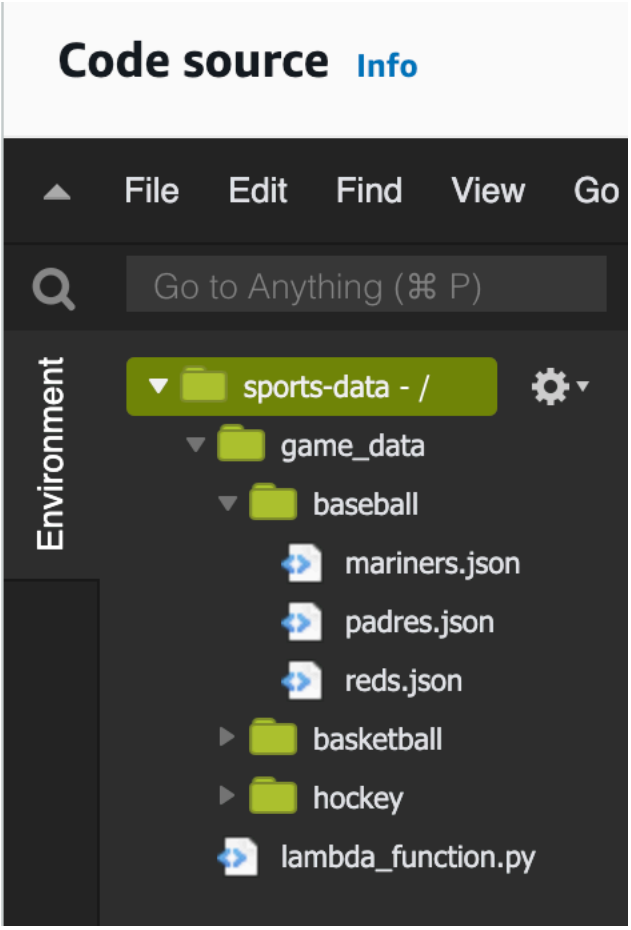

Fig. 1: folder hierarchy in AWS Lambda

Setup API

Our API is set up to use a Get Query String as an input. The mapping template for the get-integration-request is shown here:

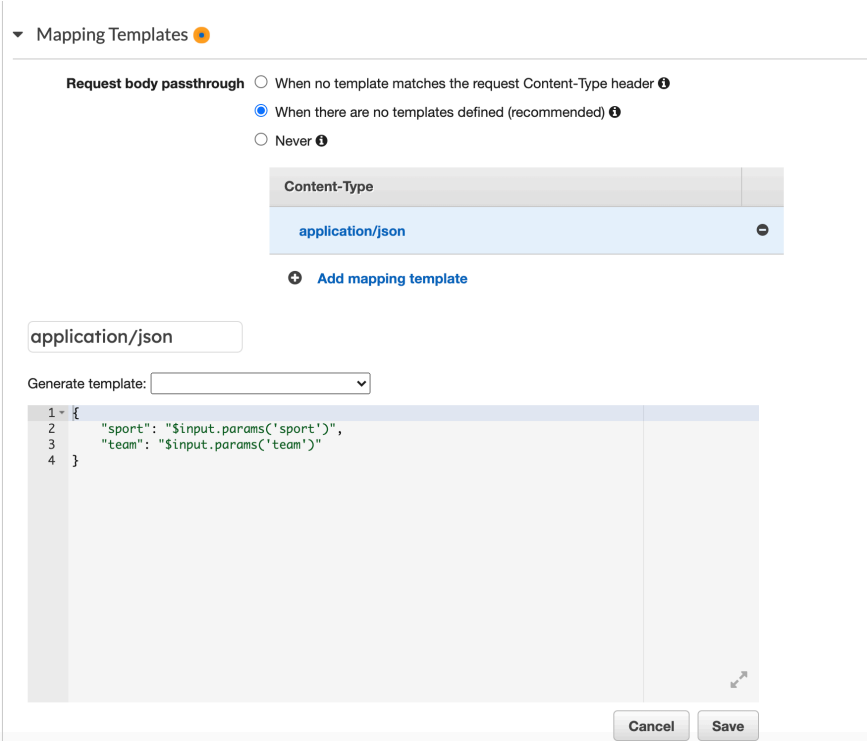

Fig. 2: Mapping template for get-integration-request

Overall, the whole API method execution is art of a deployment called `hw3at1`, as in homework3 attempt 1 (but miraculously got it on the first attempt).

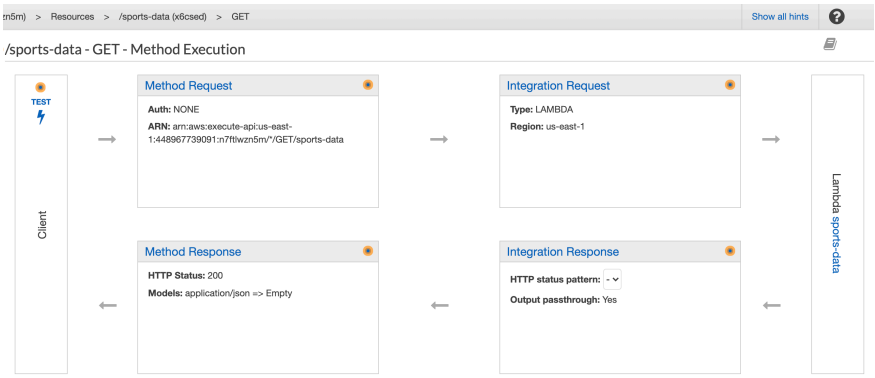

Fig. 3: sports-data - GET - method execution

Part 2: The Lambda Function

Logic

The lambda functionality should go as follows:

1. Accept two queries from API call
2. Get the sport/team data
    2.1 Check if the sport exists, if not, **return a reply statement and a list of possible sports.**
    2.2 If the sport does exists, do the same: check if the team exists, and if not, **return a reply statement and a list of possible teams.**
    2.3 If the team and sport exist, parse the corresponding JSON file for the contents and return a list of dictionaries (which is essentially what a JSON array is, right?) of the team data.
    2.4 Then, send this data to a custom print function that essentially formats the data into a format of `sport -> team -> game-data`
    2.5 The custom print function return a dictionary which is then returned to the API.

Accept Input

For accepting input, we will store the string query into two variables, `query_sport` and `query_team`:

```
query_sport = str(event['sport']).lower().strip()
query_team = str(event['team']).lower().strip()
```

Get Team Data

We'll send these variables to a function, `get_team_data`:

```
team_data = get_team_data(query_sport, query_team)
```

That function looks like this:

```
def get_team_data(query_sport, query_team):
    directory_path = "game_data/"
    there_is_data = check_if_data_exists(
        query_sport, query_team, directory_path)
    if there_is_data is True:
        directory_path += query_sport + "/" + query_team + ".json"

        with open(directory_path) as json_data:
            data = json.load(json_data)

        return list(filter(lambda x: x["game_id"] < 6, data))
    elif there_is_data[0] == "NO_SPORT":
        return ["Error.", f"Sport: {query_sport} was not found. Please try
    from one of these sports:", there_is_data[1]]
    elif there_is_data[0] == "NO_TEAM":
        return ["Error.", f"Team: {query_team} was not found. Please try from
    one of these teams:", there_is_data[1]]
```

In this function, another function, `check_if_data_exists` is called, which looks like this:

```
def check_if_data_exists(query_sport, query_team, directory_path):
    types_of_sports = [
        os.path.splitext(dir_content)[0]
        for dir_content in os.listdir(directory_path) if not
    dir_content.startswith(".")
    ]
    if query_sport in types_of_sports:
        sport_teams = [
            os.path.splitext(dir_content)[0]
            for dir_content in os.listdir(directory_path + str(query_sport))
    if not dir_content.startswith(".")
        ]
        if query_team in sport_teams:
            return True
        else:
            return ["NO_TEAM", sport_teams]
    else:
        return ["NO_SPORT", types_of_sports]
```

As you can see, if either no team or no sport exists, a list is returned, with a ‘header’ (there’s probably a proper way to do a header, and this probably isn’t it but it works!) that describes which wasn’t found and a nested list of the correct teams that we do have information on.

Give the data

If there is no team or no sport, there shouldn't be a need to call our custom print function, so before we do that, we'll catch our returned data from `get_team_data` and see if there is a "header" that calls not to send to that function. This is implemented as such:

```
def lambda_handler(event, context):

    ...

    if team_data[0] == "Error.":
        response = [
            team_data[1],
            team_data[2],
            "For more options, please consider upgrading to our Pro Plan. (1
    BTC per request)"
        ]
    else:
        response = print_team_data(query_sport, query_team, team_data)
```

`response` is what will eventually be returned, and here the `lambda\_handler`checks to see if the response should be something that is formatted by our `print\_team\_data` function. If it is, then it is sent to:

```
def print_team_data(query_sport, query_team, team_data):
    game_log = {
        query_sport: {
            query_team: []
        }
    }
    for game in team_data:
        response = ""
        response += f"{game['date']} - A {query_team.capitalize()}"

        if game['game_won']:
            response += " win against "
        elif not game['game_won']:
            response += " loss to "

        response += f"the {game['opponent']}. The score was
    {game['points_for']}-{game['points_against']}."

        game_log[query_sport][query_team].append(response)
    return game_log
```

Here, we're creating a dictionary that has a nested dictionary with a list in it. As the `for`loop parses through the data, it formats a response to look like this:

Nov 27, 2021 - A Blues win against the Columbus Blue Jackets. The score was 6-3.

Here's what it looks like in actuality:

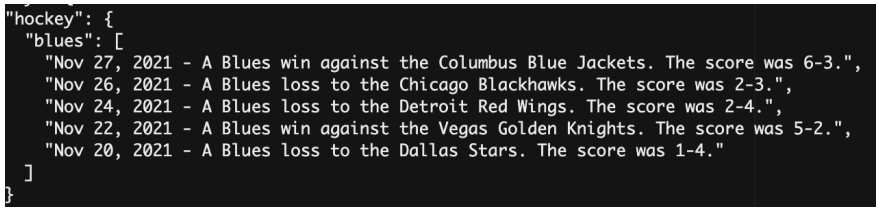

Fig. 4: Sample response from API call

Returning to API

Once that is formatted, the `lambda\_handler` returns a standard RestAPI response, with a `statusCode`, `body`,and `headers`.

```
def lambda_handler(event, context):

    ...

    return {
        'statusCode': 200,
        'body': response,
        'headers': {
            'Content-Type': 'application/json',
        },
    }
```

Running the API

I'm using Paw on macOS to run some API calls. To test is quite simple, create a new `get` request with our API link: `[https://n7ftlwzn5m.execute-api.us-east-1.amazonaws.com/default/sports-data]`

Then we select URL Params and enter each parameter and value. Let's try one for `hockey` and `blues`:

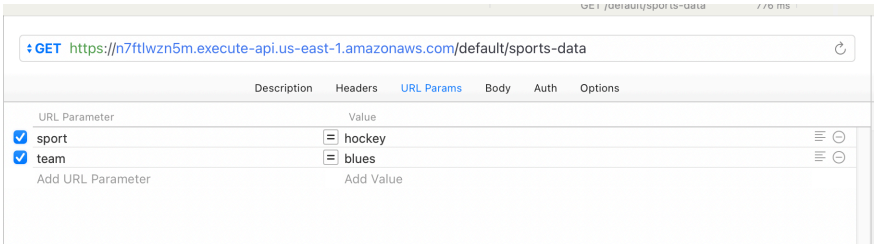

Fig. 5: Paw API request setup for hockey

Our response worked! Let's see the result:

Fig. 6: Paw API response for hockey

We see our data formatted as intended with the correct information (Let's Go Blues!)

Let's try another request, but this time for baseball and the Cincinnati Reds

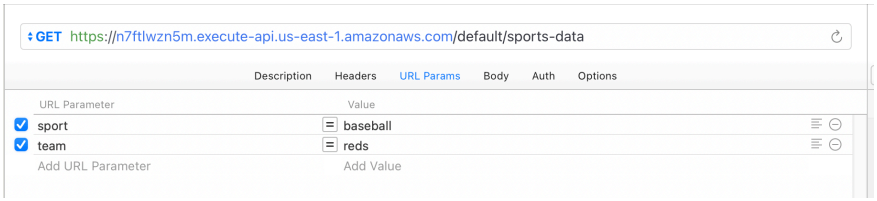

Fig. 7: Paw API request setup for baseball

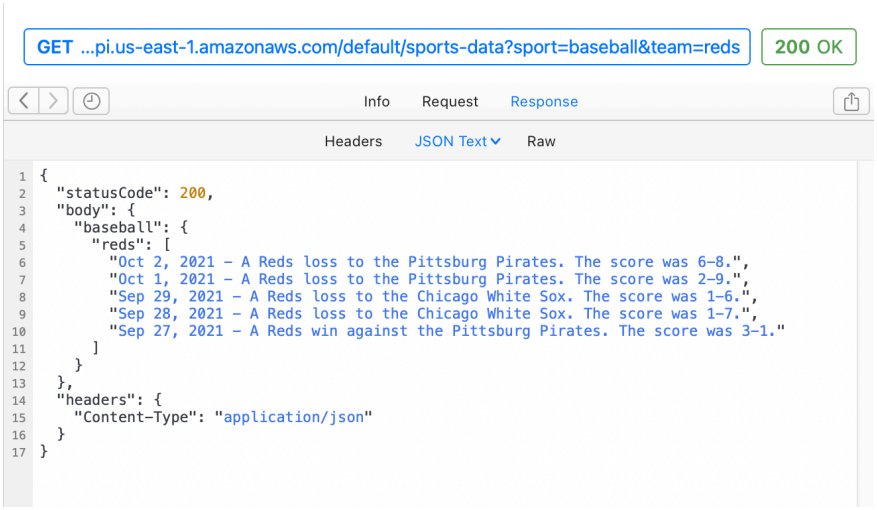

Fig. 8: Paw API response for baseball

Let's make sure that it responds as intended when we query information that is not in our dataset. I'm going to do basketball and the Harlem Globetrotters:

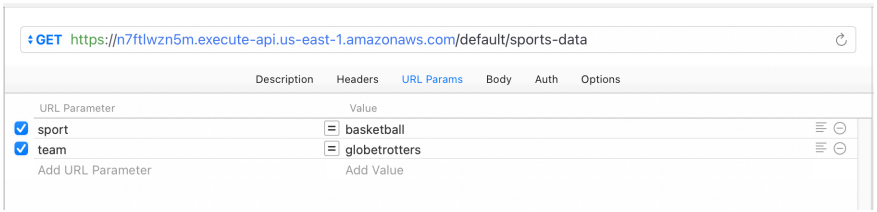

Fig. 9: Paw API request setup for basketball

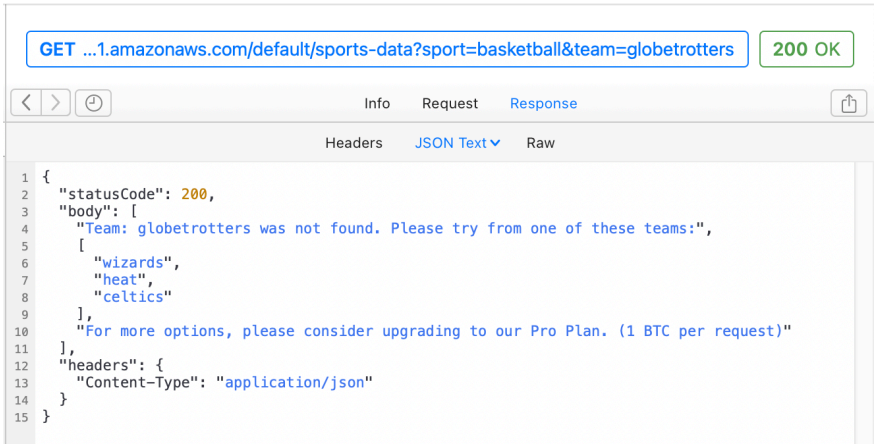

Fig. 10: Paw API response for basketball

The API and Lambda work as intended, and can be accessed at:

https://n7ftlwzn5m.execute-api.us-east-1.amazonaws.com/default/sports-data