Game Database

2018 January

A search front end for metacritic.com

**DynamoDB Version**



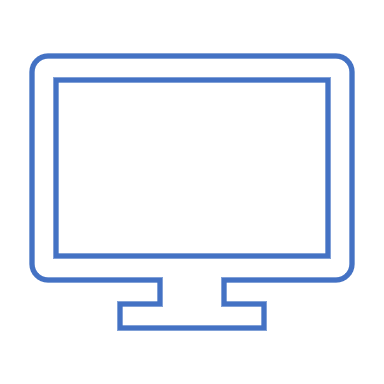
**RDS Version**



# Overview

This project downloads video game data from metacritic.com and uses it to host a searchable front end, using a server-less strategy.

**Platforms:** Python 3, AWS, MySQL, DynamoDB, Lambda



λ

Source: /web\_page /lambda /databases

**#1a. Downloading game data from metacritic.com**

This is done by the "web\_scrape.py" script.

Example usage:

python3 web\_scrape.py louis2018 data002\_004.tsv 2 4

This will download data from web pages #2 to #4. The data will go into AWS S3 bucket "louis2018", key name "data002\_004.tsv".

The very first web page is 0. As of early January, 2018, the very last website is 227.

The design of the script allows multiple EC2 instances to be used to process the metacritic.com data. One EC2 instance can download from web page 0 through 99, while a second EC2 instance can download from web page 100 through 199.

**#1b. Download the data in S3 to the ".\data" directory**

The script in step #1a downloads data to a S3 bucket.

The scripts in step #2 assumes the data is in a ".\data" directory.

One way is to use:

import boto3

bucket\_name = "louis2018"

bucket = boto3.resource('s3').Bucket(bucket\_name)

s3 = boto3.client('s3')

for file in bucket.objects.all():

if file.key.endswith(".tsv"):

s3.download\_file(bucket\_name, file.key, file.key)

**#2. Setting up databases**

To better learn about database programming, this project uses two different databases - MySQL and DynamoDB.

**AWS RDS MySQL**

Use the AWS web console to create an instance.

Use "setup\_rds.sql" to create the tables and indexes.

Use "upload\_data\_to\_rds.py" to upload the data to AWS RDS.

Near the top of the code is a block about database connection that needs to be modified:

###############################################

# TODO - Database connection settings

connection = pymysql.connect(host='localhost',

user='root',

password='123456',

db='mydb')

###############################################

Run the code without argument.

This code assumes the data is in the ".\data" directory.

If using the project's dataset, after running the script the table sizes will be:

select count(id) from Games; -- 4441

select count(game\_id) from GameGenre; -- 6643

**AWS DynamoDB**

The file "setup\_dynamoDB.py" creates the table.

Near the top of the code is database connection information that needs to be customized:

###############################################

# TODO - Database connection settings

dynamodb = boto3.resource('dynamodb', region\_name='us-west-1')

###############################################

Run the code without argument.

Check via web console to confirm the table creation.

The file "upload\_data\_to\_dynamoDB.py" uploads the data.

Near the top of the code is database connection information that needs to be customized:

###############################################

# TODO - Database connection settings

dynamodb = boto3.resource('dynamodb', region\_name='us-west-1')

###############################################

Run the code without argument.

This code assumes the data is in the ".\data" directory.

During the data creation step, the write capacity is set at "10". To make the upload process faster, manually change this to a higher number, say 200.

If using the project's dataset, after running the script the DynamoDB "Games" table will contain 6643 items, taking up 1.12 MB.

**#3. Querying the Databases**

**AWS RDS MySQL**

The Lambda handler code is in "lambda/query\_RDS.py".

The code uses the "pymysql" package. This means two additional directories, "pymysql" and "PyMySQL-0.8.0.dist-info", needs to be uploaded along with "query\_RDS.py".

You get these two directories using:

pip install pymysql -t ./

**DynamoDB**

The Lambda handler code is in "lambda/query\_dynamoDB.py".

**#4. Web Page Front End**

The files are in the "web\_page" directory.

The RDS has more capability than the DynamoDB, so there's a separate web page for each database.

At the top of the "search\_RDS.js" and "search\_dynamoDB.js":

// TODO - modify API url as needed

var api\_url = "https://f0qj73614m.execute-api.us-west-1.amazonaws.com/beta/query-rds-db"

# #1. Downloading Game Data

Game data is downloaded from <http://www.metacritic.com/> into a tab delimited format, using the "**web\_scrape.py**" script.

## Extracting data for a specific game

This is done using the class "GameInfoParser".

Example URL: <http://www.metacritic.com/game/pc/civilization-ii>

The relevant HTML:

<span class="data" itemprop="datePublished">Feb 29, 1996</span>  
The "datePublished" is unique

<div class="label">User Score</div>  
<div class="metascore\_w user large game positive">9.0</div>  
The "metascore\_w user large game positive" is not reliable. Instead, rely on the preceding "<div ...> User Score" tag.

The above pattern happens two times per page, and the usually have the same score. However, some game pages seem to have glitches, where the two patterns have different scores. The first patter appears to be the accurate one.

<a href="/game/pc/civilization-ii/user-reviews">370 Ratings</a>  
The "/game/pc/civilization-ii/user-reviews" is not unique. The data node needs to end with "Ratings".

The state machine:

Starting state = 0.

|  |  |  |
| --- | --- | --- |
| Current State | Condition / Action | Next State |
| 0 | Attribute is itemprop="datePublished" | 1 |
| 1 | Record the data node as "release\_date". | 0 |
| 0 | Data node = "User Score" | 2 |
| 2 | <div> tag | 3 |
| 3 | Record the data node as "user\_score". Do this only once. | 0 |
| 0 | Attribute is href="/game/pc/civilization-ii/user-reviews" | 4 |
| 4 | Check that the data node has two words and ends in "Ratings". Record the first word as "num\_user\_ratings". | 0 |

Processing the webpage for a game is implemented in class "GameInfoParser".

Testing the "GameInfoParser":

import requests

from html.parser import HTMLParser

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0'}

response = requests.get("http://www.metacritic.com/game/pc/civilization-ii", headers=headers)

parser = GameInfoParser()

parser.feed(response.text)

print(parser.release\_date)

print(parser.user\_score)

print(parser.num\_user\_ratings)

## Extracting the list of games

This is done using the class "GameListParser".

Example URL: <http://www.metacritic.com/browse/games/genre/metascore/strategy/pc?view=condensed&page=0>

The relevant HTML:

<div class="product\_condensed">

<li class="product game\_product first\_product">  
<a href="/game/pc/command-conquer">Command &amp; Conquer</a>  
<div class="metascore\_w small game positive">94</div>

<li class="product game\_product">

The state machine:

Starting state = 0.

|  |  |  |
| --- | --- | --- |
| Current State | Condition / Action | Next State |
| 0 | A <div> tag that has class="product\_condensed". | 1 |
| 1 | A <li> tag, with a "class" attribute that contains "game\_product". | 2 |
| 2 | An <a> tag. Extract the "href" attribute to "url". | 3 |
| 3 | Extract the data node to "title". | 4 |
| 4 | A <div> tag, with a "class" attribute that contains "metascore\_w" | 5 |
| 5 | Extract the data node to "metascore" | 1 |

Processing a webpage that contains a list of games is implemented in class "GameListParser".

Testing the "GameListParser":

import requests

from html.parser import HTMLParser

headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0'}

response = requests.get("http://www.metacritic.com/browse/games/genre/metascore/strategy/pc?view=condensed&page=0", headers=headers)

parser = GameListParser()

parser.feed(response.text)

print(len(parser.url), parser.url[2])

print(len(parser.title), parser.title[2])

print(len(parser.metascore), parser.metascore[2])

## Breaking up the list of websites to be downloaded

This is done by the class "WebPages\_List".

The goal is to download the information using multiple AWS EC2 instances. As of January 6, 2018, there are 228 websites to examine using the "GameListParser" class.

For example, one EC2 instance can work on downloading website 0 through 49, while a second one can work on website 50 through 99.

The class "WebPages\_List" will generate a URL for a given integer between 0 and 227.

Test code:

web\_pages\_list = WebPages\_List()

web\_pages\_list.length() # 228

web\_pages\_list.get\_task(0) # ('strategy', http://www...page=0)

web\_pages\_list.get\_task(34) # ('strategy', http://www...page=34)

web\_pages\_list.get\_task(35) # ('action', http://www...page=0)

web\_pages\_list.get\_task(35 + 48) # ('action', http://www...page=48)

web\_pages\_list.get\_task(227) # ('turn-based strategy', http://www...page=9)

web\_pages\_list.get\_task(227 - 10) # ('third-person shooter', http://www...page=9)

## Running the script

python3 web\_scrape.py louis2018 data002\_004.tsv 2 4

This will download web sites #2 to #4. The data will go into AWS S3 bucket "louis2018", key name "data002\_004.tsv".

## Errors Found

### Broken Links

The script checks for "release\_date" and "user\_score" == "unknown". Sometimes this is due to the http request to grab the web page failing to complete. Retrying the HTTP fixes this situation.

Another situation is that the link is actually broken.

As of early January 2018, the following broken links have been found:

<http://www.metacritic.com/browse/games/genre/date/action/pc?page=11>  
Warlocks vs Shadows - supposedly released Aug 19, 2015, with metascore of 68

<http://www.metacritic.com/browse/games/genre/date/adventure/pc?view=condensed&page=5>   
NightCry - supposedly released Mar 28, 2016, with metascore of 46

<http://www.metacritic.com/browse/games/genre/date/adventure/pc?view=condensed&page=22>  
The Messenger - supposedly released Feb 14, 2001, with metascore of 61

### Duplication

**Star Wars Battlefront II**

"Star Wars Battlefront II" is a 2017 game, yet it's also available in 2005:

<http://www.metacritic.com/browse/games/genre/date/action/pc?page=39>

This created a duplicated entry in the database. The wrong entry has a colon in the game name. So "Star Wars Battlefront II" is the correct data, while "Star Wars**:** Battlefront II" is the wrong data.

The issue has been manually fixed in the database:

select \* from Games

where title="Star Wars: Battlefront II"; -- id 2222

delete from GameGenre

where game\_id=2222;

delete from Games

where id=2222;

# #2. Setting up Databases

## Data File Format

The data generated by "web\_scrape.py" is in tab delimited format. Each row is:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| genre | title | url | meta score | release date | user score | number of  user ratings |
| Strategy | Tiny Metal | http://... | 71 | Dec 21, 2017 | tbd | 11 |

## Database Storage Format

**Non-numeric scores are stored as -1**

Titles that did into catch on, or titles that have been recently released, might have a score of "tbd".

In the database these scores will be encoded as -1.

**Multiple Genres**

Games can belong to multiple genres.

### RDS Storage Format

The data is stored as two tables.

**Games Table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| id | title | url | release\_date | metascore | user\_score | num\_user\_ratings |
| 1 | Tiny Metal | http://... | Dec 21, 2017 | 71 | tbd | 11 |

The combination of "title" and "release\_date" should be unique.

**GameGenre Table**

A separate "GameGenre" table to support multiple genres per game.

|  |  |
| --- | --- |
| game\_id | genre\_id |
| 1 | 12 |

Genre ID Encoding

|  |  |
| --- | --- |
| 1 | action |
| 2 | adventure |
| 3 | first person shooters |
| 4 | flight |
| 5 | platformer |
| 6 | puzzle |
| 7 | racing |
| 8 | real-time strategy |
| 9 | role-playing games |
| 10 | simulation |
| 11 | sports |
| 12 | strategy |
| 13 | third-person shooter |
| 14 | turn-based strategy |

Indexing

|  |  |
| --- | --- |
| genre\_id | To be able to narrow down the search by genre. |
| release\_date | To be able to narrow down the search by year. |

### DynamoDB Storage Format

**Games table**

**Secondary Indexes**

|  |  |
| --- | --- |
| {genre\_year, metascore} | Index "sort\_by\_metascore" - to retrieve data of a specific genre and year, sorted by "metascore". |
| {genre\_year, user\_scorex100} | Index "sort\_by\_user\_score" - to retrieve results sorted by user score. |
| {genre\_year, num\_user\_ratings} | Index "sort\_by\_num\_user\_ratings" - to retrieve results sorted by user ratings. |

The "genre\_year" is a partition key computed using:

genre\_year = (release year) \* 100 + genre\_id

**Primary Index**

The secondary indices are for retrieving the data, but they are not necessarily unique - for example, both "Block'hood" and "Warhammer 40,000: Dawn of War III" are released in the same year (2017), belong to the same genre (strategy), and have the same metascore (77).

|  |  |
| --- | --- |
| {genre\_year, title} | A unique index to satisfy the primary key requirement. |

Note that all indexes have the same partition key of "genre\_year".

**Declared columns**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **genre\_year** | **title** | **metascore** | **user\_scorex100** | **num\_user\_ratings** |
| number | string | number | number  (user score of 3.3  stored as 330) | number |

**Undeclared columns**

url, release\_date

# #3. Querying the Databases

## AWS RDS MySQL

**Request format**

{

"genre\_id": "12",

"start\_year": "2013",

"end\_year": "2017",

"sort\_by": "metascore",

# "sort\_by": "user\_score",

# "sort\_by": "num\_user\_ratings",

}

**Testing - standard case**

import requests, json

url = "https://f0qj73614m.execute-api.us-west-1.amazonaws.com/beta/query-rds-db"

data = {

"genre\_id": "12",

"start\_year": "2013",

"end\_year": "2017",

"sort\_by": "metascore",

# "sort\_by": "user\_score",

# "sort\_by": "num\_user\_ratings",

}

data\_str = json.dumps(data)

r = requests.post(url, data=data\_str)

response\_data = json.loads(r.text)

response\_data

**Testing - error case**

data = {

"genre\_id": "12",

"start\_year": "2013x",

"end\_year": "2017",

"sort\_by": "metascore",

# "sort\_by": "user\_score",

# "sort\_by": "num\_user\_ratings",

}

## DynamoDB

**Request format**

{

"genre\_id": 12,

"year": 2016,

"sort\_by": "metascore",

# "sort\_by": "user\_scorex100",

# "sort\_by": "num\_user\_ratings",

}

**Testing - standard case**

import requests

import json

url = "https://f0qj73614m.execute-api.us-west-1.amazonaws.com/beta/query-dynamo-db"

data = {

"genre\_id": "12",

"year": "2016",

"sort\_by": "metascore",

# "sort\_by": "user\_scorex100",

# "sort\_by": "num\_user\_ratings",

}

data\_str = json.dumps(data)

r = requests.post(url, data=data\_str)

response\_data = json.loads(r.text)

**Error case**

data = {

"genre\_id": "12",

"year": "2016",

"sort\_by": "metascorex",

# "sort\_by": "user\_scorex100",

# "sort\_by": "num\_user\_ratings",

}

# #4. Web Page Front End

Clicking on the "search" button can start the search process:

search\_button\_click\_handler()  
 |🡪 compose\_search\_request()  
 |🡪 render\_search\_results(results, sort\_by)  
 |🡪 create\_tag(tag\_name)  
 |🡪 create\_tag\_with\_text(tag\_name, text, class\_name = null)  
 |🡪 create\_tag\_with\_link(tag\_name, text, url)  
 |🡪 render\_error(error)

User interface changes also starts the search process:

|  |
| --- |
| // user pressing "enter" in the "year" textbox will start the search process  Year\_tb\_keyup\_handler(e)  // user changing the ratings method will start the search process  Ratings\_cb\_change\_handler()  // user changing the genre selection will start the search process  Strategy\_cb\_change\_handler() |

|  
 |🡪 search\_button\_click\_handler()