Assignment 1: Web Development

Due Oct 17, 2022 by 12p.m. **Points** 23 **Submitting** a file upload
Available Sep 14, 2022 at 6p.m. - Oct 19, 2022 at 12p.m.

This assignment was locked Oct 19, 2022 at 12p.m..

Objective

This assignment will provide you with experience developing a storage web application with an in-memory key-value memory cache. You will be using Python and the Flask framework to implement your application. You will also get experience deploying and running your application on Amazon EC2.

You do not need to develop your web application on EC2, instead you may develop locally on your own laptops and share your web application sources among your team mates via GitHub. Later, when you are ready to deploy, you will have to move your web application to EC2. Key concepts required for this assignment will be covered in lectures, potential tutorials and exercises. First, understand some of the key concepts, then, design your solutions, finally, develop, test, and deploy. All these tasks should be shared across the team you are working in and can be divided among your team mates. We want you to plan, not just blindly dive into development.

Note: when you're ready to get started on A1, see Set-Up and Resources (https://q.utoronto.ca/courses/285775/pages/set-up-and-resources) .

Web Application Details & Description

The application consists of 5 key components:

1. The web browser that initiates requests,
2. The web front end that manages requests and operations,
3. The local file system where all data is stored,
4. The mem-cache that provides faster access, and
5. The relational database (RDBMS), which stores a list of known keys, the configuration parameters, and other important values.

You should implement the following components:

1. A web front end that provides the following functionalities:
    1. A page to upload a new pair of key and image: the key should be used to uniquely identify its image. A subsequent upload with the same key will replace the image stored previously. The web front end should store the image in the local file system, and add the key to the list of known keys in the database. Upon an update, the mem-cache entry with this key should be invalidated.

2. A page that shows an image associated with a given key.
3. A page that displays all the available keys stored in the database.
4. A page to configure the mem-cache parameters (e.g., capacity in MB, replacement policy) as well as clear the cache.
5. Display a page with the current statistics for the mem-cache over the past 10 minutes.

2. Key-Value Memory Cache: a mem-cache is an in-memory cache that should be implemented as a Flask-based application. The mem-cache should be able to support these operations:
   A. PUT(key, value) to set the key and value (contents of the image).
   B. GET(key) to get the content associated with the key.
   C. CLEAR() to drop all keys and values.
   D. invalidateKey(key) to drop a specific key.
   E. refreshConfiguration() to read mem-cache related details from the database and reconfigure it based on the values set by the user (see 4. above).

The mem-cache should support two cache replacement policies:

A. **Random Replacement**: Randomly selects a key and discards it to make space when necessary. This algorithm does not require keeping any information about the access history.
B. **Least Recently Used**: Discards the least recently used keys first. This algorithm requires keeping track of what was used when, if one wants to make sure the algorithm always discards the least recently used key.

The mem-cache and the web front end should be able to communicate with the database. The mem-cache uses the database to read configuration parameters (capacity in MB, replacement policy, etc.) and to store statistics (number of items in cache, total size of items in cache, number of requests served, miss rate and hit rate). The mem-cache should store its statistics every 5 seconds. The web front end uses the database to keep track of the list of known keys, to set configuration values for the mem-cache and to query statistics stored by the mem-cache. Figure 1 shows the workflow of a data GET request that is not on the mem-cache, that is, a cache miss. In this case, after receiving the request from the web browser, the web front end sends a GET request to the mem-cache (step 1). However, there is no required data to be retrieved, so the mem-cache signals back a MISS (step 2). The web front end then reads the data from the local file system (steps 3 and 4), and updates the mem-cache to include the most recent entry (step 5 and 6). You should implement two replacement policies for the mem-cache: Random and Least-Recently-Used (LRU).

Similarly, Figure 2 shows the workflow of a PUT request: the web browser sends the request to the web front end, which passes the data to the local file system to be written (step 1). Due to the data change, the web front end asks the mem-cache to invalidate the corresponding entry (step 2). Once this is done, the mem-cache sends back an OK acknowledgement to the web front end (step 3), which completes the operation.

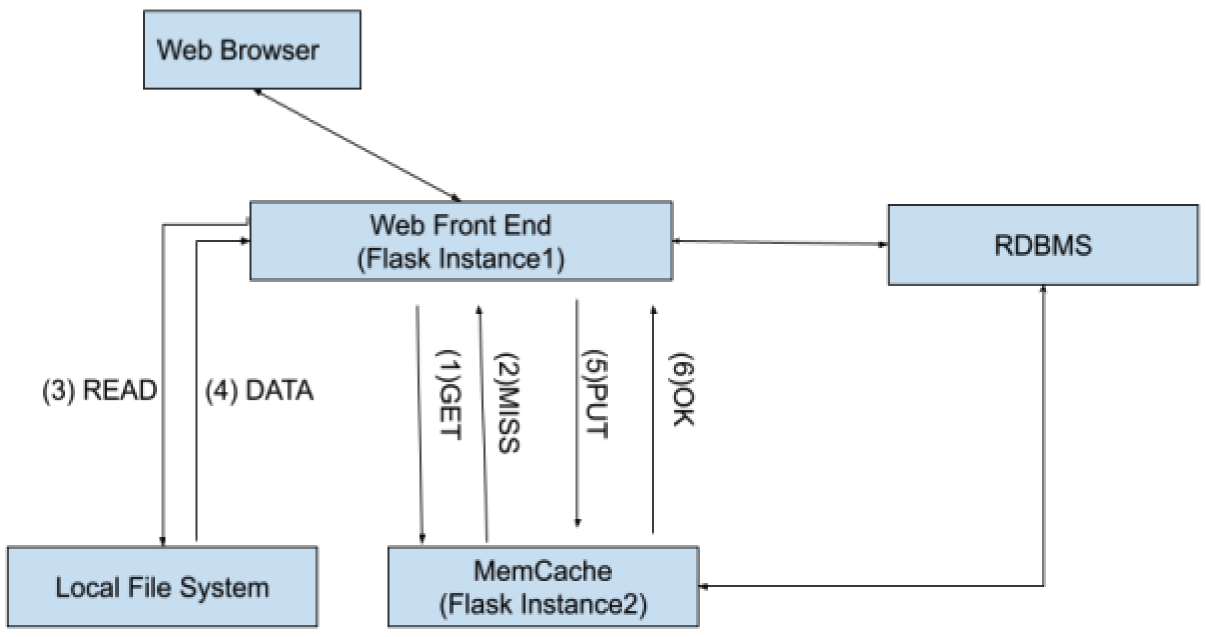

Fig 1. Workflow of a data GET request with a cache miss

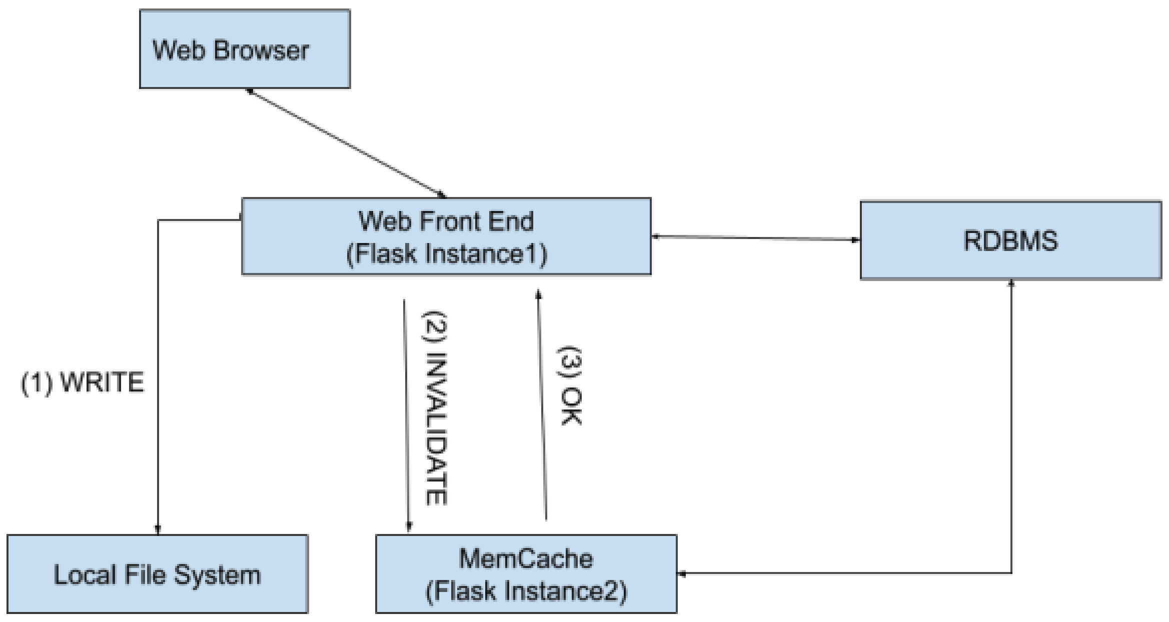

Fig 2. Workflow of a PUT request

Requirements

1. Implement all the required components of the application system. Starter code for a Flask-based mem-cache is available in **Set-Up & Resources (https://q.utoronto.ca/courses/285775/pages/set-up-**

and-resources). It uses a global variable to store data in memory as a Python dictionary and includes two endpoints (get, put) that return json.

2. You must create a private GitHub repository within your team and do all development in this repository; it is recommended to use **best practices** (https://docs.github.ncsu.edu/github-best-practices/).
3. All files should be stored in the local file system (i.e., on the virtual hard drive of the EC2 instance).
4. The keys of all files and their location on the local file system should be stored in the database. Do not store the files themselves in the database. It is up to you to design the database schema, but make sure that you follow design practices and that your database schema is properly normalized.
5. Cache metadata such as miss rate information, utilization over time, or configuration parameters should also be stored in a relational database.
6. To allow for external testing by an independent party, your application should (in addition to your web interface) include three URL endpoints to automatically upload files and retrieve files. Imagine that a testing script would send HTTP requests to your URL endpoints with the parameters in the body. Your implementation should not accept HTTP requests where parameters have been appended to the URL. See this link for more information about how HTTP POST requests work: **https://www.w3schools.com/tags/ref_httpmethods.asp.** (https://www.w3schools.com/tags/ref_httpmethods.asp) When we conduct the demo of your web application, we may run scripts to validate that this functionality is properly implemented.
7. Your web application should run on port 5000 and be accessible from outside the instance. So, make sure that you open this port on your EC2 instance. Documentation about how you can open the port can be found **here** (https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/authorizing-access-to-an-instance.html). It is preferred to use gunicorn or uwsgi to start your Flask web application.
8. Write a shell or bash script named **start.sh** that initializes your web application to ease bring your web application up and make it ready to run.
9. The two automatic testing endpoints should conform to the following interfaces:

Upload:

```
relative URL = /api/upload
enctype = multipart/form-data
method = POST
POST parameter: name = key, type = string
POST parameter: name = file, type = file
```

Retrieve all keys:

```
relative URL = /api/list_keys
```

```
method = POST
```

Retrieve an image associated with a specific key:

```
relative URL = /api/key/<key_value>
method = POST
```

These endpoints should generate responses in the following JSON format:

In case of success for the **upload** interface:

```
{
    "success": "true"
}
```

In case of success for the **retrieve** interface while fetching all keys:

```
{
    "success": "true",
    "keys": [Array of keys(strings)]
}
```

In case of success for the **retrieve** interface while fetching a particular key:

```
{
    "success": "true",
    “content” : file contents
}
```

In case of failure of any of the calls of the **upload** or **retrieve** interface:

```
{
    "success": "false",
    "error": {
        "code": servererrorcode
        "message": errormessage
     }
}
```

Think about possible failure scenarios and provide meaningful error messages as output.

Report

A written report is required alongside the technical requirements of this assignment. The specification and contents of the report are as follows:

- One report per group.
- Group number, assignment number, each member's name and student ID required at the top of page one.
- Include page numbers on the bottom right corner of each page (including the first).
- A page limit of 5 pages for the report (additional pages will result in deducted marks); here, less is more if you are able to succinctly describe everything.
- Good formatting & legibility (Click here (https://guides.lib.berkeley.edu/how-to-write-good-documentation) for tips on how to write documentation).
- Include the following sections
  - **General Architecture** of your application (use figures and diagrams as needed);
  - **Database Schema:** use ER diagrams to describe your schema;
  - **Design Decisions**: what the key decision(s) were in designing your web application, any alternatives you considered, and why you made your choice.
  - **Performance Graphs**:
    - 3 groups of graphs: group 1 with a 20:80 read/write ratio, group 2 with a 50:50 read/write ratio, group 3 with an 80:20 read/write ratio. Each group will include:
      - A latency graph with varying number of requests on the x axis, starting with 1 request, incrementing up to the number your system can handle (chose a reasonable increment). Show latency for each caching policy (no cache, random replacement, and least recently used) on your graph, labeled.
      - A throughput graph that shows maximum requests that can be processed per varying units of time (at your discretion). Include throughput for each caching policy (no cache, random replacement, and least recently used) on your graph, labeled.
    - In total, you should have 6 graphs - you could lay the graphs out in three rows of two graphs each (aim for legibility and keep legends and any text in the graph large enough); you may use other layouts.
    - Include a brief explanation of why the results are the way they are.

Note: when you're ready to submit, see Submission & Demo Instructions (https://q.utoronto.ca/courses/285775/pages/submission-and-demo-instructions) .

Document Revisions

Here, we manually track major revisions to this document that occurred after the document was released.

Date	Revision

Sep 20th, 2022	Item 6 in the initial list describing components to implement was at the wrong indentation (it now became Item 2 in the outer list.)
Sep 20th, 2022	Added this Document Revision section

Assignment 1 Rubric

Criteria	Ratings		Pts
UI/UX Being able to navigate through all pages easily (using sensible menus and buttons), properly showing the files if fetched. While 1779 is not a design course, it is expected that your application will include reasonable styling to provide a pleasant user experience.	**3 pts** **Full Marks**	**0 pts** **No Marks**	3 pts
Functionality Implementation of features listed in the Web Application Details section including proper database design.	**10 pts** **Full Marks**	**0 pts** **No Marks**	10 pts
Testing Interface The web application is subjected to a number of requests from an independent party. Requests expecting success and failure will be selected. The number of requests correctly processed over all requests submitted assess the outcome of this criterion.	**10 pts** **Full Marks**	**0 pts** **No Marks**	10 pts
Documentation All code should be properly formatted and reasonably documented. The overall architecture of the application and how different elements of the application are connected to each other should be described. The database schema should be described. Important design decisions should be reviewed and alternatives given; a brief justification should be provided for why which decision was taken. Keeping the design simple is reasonable. Measurement methodology and workload are described. All performance graphs specified in the assignment handout are provided, labeled and legible. An explanation of why the results are what they are is provided. See Report Submission page for rubric and mark allocation.	**0 pts** **Full Marks**	**0 pts** **No Marks**	0 pts
			Total Points: 23