ECE1779: Introduction to Cloud Computing

Winter 2022

Assignment 1
Web Development

Due Date

Feb 21, 2022

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.

Web Application Details

The application consists of 5 key components: the **web browser** that initiates requests, the **web front end** that manages requests and operations, the **local file system** where all data is stored, the **mem-cache** that provides faster access, and 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:
    a. 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.
    b. A page that shows an image associated with a given key.
    c. A page that displays all the available keys stored in the database.
    d. A section to configure the mem-cache parameters (e.g. capacity in MB, replace policy).
    e. 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 with default values

    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.

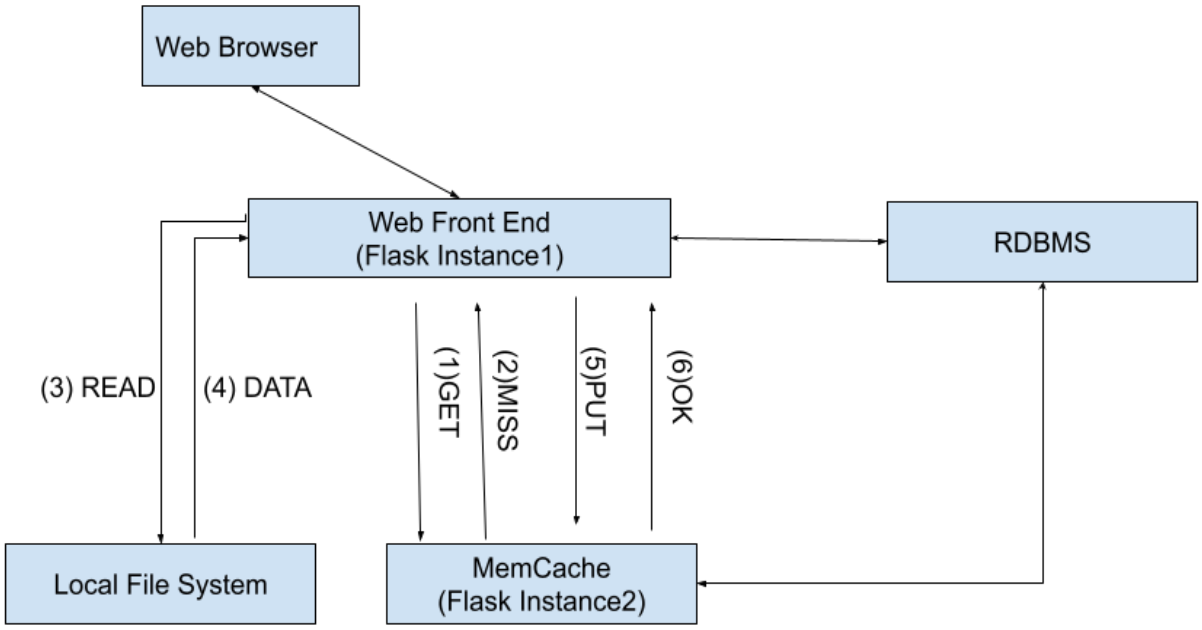

Fig 1. Workflow of a data GET request with a cache miss

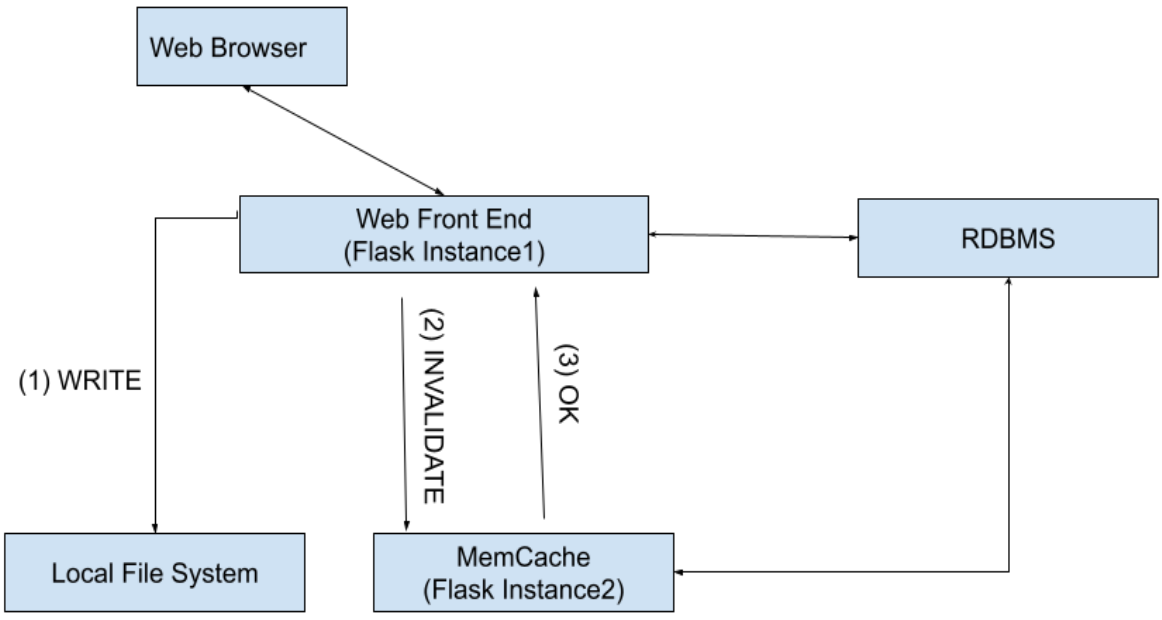

Fig 2. Workflow of a PUT request

Requirements

1. Implement all the required components of the application system. Starter code for a Flask-based mem-cache is available here. It uses a global variable to store data in memory as a Python dictionary and includes two endpoints (get, put) that return json.
2. All files should be stored in the local file system (i.e., on the virtual hard drive of the EC2 instance).
3. 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. The admins should be able to see the mem-cache parameters.
4. Cache metadata such as miss rate information, utilization over time, or configuration parameters should also be stored in a relational database.
5. To allow for automatic testing, your application should **(in addition to your web interface)** include three URL endpoints to automatically upload files and retrieve files. Our automatic testing script will 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
6. 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 failure:

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

     ```
     {
     ```

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

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

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

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

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

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

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

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

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

Deliverables

We will test your application using your AWS account. For this purpose, your application should be pre-loaded on an EC2 instance. Please **suspend** the instance when you submit your project to prevent charges from occurring while the TA gets around to grading your submission. Make sure to not restart the instance from the moment you submit your project.

You should write a shell or bash script called **start.sh** that initializes your web application. This script should be put in the home directory of the user associated with the EC2 keypair (e.g., /home/ubuntu). 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.

It is preferred to use *gunicorn* or *uwsgi* to start your Flask web application.

Submit the assignment only once per group. Clearly identify the names and student IDs of group members in the documentation.

To submit your project, upload to Quercus a single tar file with the following information:

1. Application documentation in a **PDF** file (documentation.pdf). Include a description of how to use your web application as well as the general architecture of your application (using figures and diagrams if needed). Also include a figure describing your database schema. Click here for tips on how to write documentation. Your documentation will be marked based on how cohesive it is and how well you are able to explain the technical details of your implementation.
2. In addition to the documentation, put the first name, last name, and student ID of each student in a separate line in a text file named group.txt. Please make sure that the first and last names in this group.txt file match exactly how your name is displayed in Quercus.
3. Include your AWS credentials in a text file (named credentials.txt). We will need these to log into your account. You can create credentials with limited privileges using AWS IAM if you don't want to share your password with the TA; however, make sure that you include permissions to start and stop EC2 instances. Test the credential to make sure they work.
4. Key-pair used to ssh into the EC2 instance (named keypair.pem).

   Do not forget to send the .pem files required for ssh-ing into your VM's.

5. Anything else you think is needed to understand your code and how it works.

Marking Scheme (20 points)

1. UI/UX: being able to navigate through all pages easily (using sensible menus and buttons), properly showing the files if fetched. While this is not a design course, it is expected that you application will include reasonable styling to provide a pleasant user experience (**3 points)**
2. Functionality: implementation of features listed in the Web Application Details section including proper database design (**10 points**)
3. Testing interface: our automatic testing requires necessary APIs (explained above) to function **(3 points)**
4. Documentation: all codes should be properly formatted and documented, explanations about how to log in to your amazon account, how to start the instance and how to run

the codes should be included. You should also write about the general architecture of your codes and how different elements of your codes are connected to each other. Do not forget about the database schema and group members **(4 points)**

Note that you might lose points if the TA is unable to test your application (for example, if the API endpoints do not work properly, your start.sh does not work, you sent the wrong keypair.pem, or you forgot to open required ports on your EC2 instance). Make sure that everything is in place.

Resources

Amazon provides limited free access to its infrastructure to new users through its free tier. Information about the AWS free tier, including how to apply for an account is available at https://aws.amazon.com/free. You should complete this assignment using exclusively resources from the free tier.

Important note: You AWS account is fully functional which means that it will incur charges if you use resources that do not qualify for the free tier, or if you use free-tier qualifying resources beyond the quota that Amazon provides (e.g., a 750 hour per month of EC2 t2.micro, 30 GB of EBS). You will be responsible for any charges. To avoid incurring charges make sure to use only resources from the free tier and remember to suspend your EC2 instances when not in active use.

The following video tutorials show how to :

- Create an EC2 instance
- Connect to the new instance using ssh
- Suspend an instance
- Database and AWS Flask examples covered in lectures
- Flask
- Gunicorn:
    - This is a high performance server for Python-based applications. For an example of how to run it, look at the run.sh file inside /home/ubuntu/extras/run.sh