Lab Assignment 6 (CS328 only): A url-shortening service

In this assignment you will be creating a small url-shortening and bookmarking service. The idea of this service is as follows:

- Users can create "buckets". Buckets are identified by either a custom-provided character string (if the user provides one when they create them) or a random 6-character string like 4A23GG. Buckets can also have a description of what kinds of "links" they are meant to contain.
- A bucket is a container for links. Users can add links to a bucket, and either provide a shortcut for them or have a randomly generated shortcut. For example, if a user has a bucket with id 4A23GG, then they can ask to add to this bucket the link https://en.wikipedia.org/wiki/List_of_mountains_by_elevation with "name" wikiMountains. Then someone accessing the site <ourService>/4A23GG/wikiMountains will be *redirected* to that wikipedia page.

Submission information

You can get the start files for this project by "cloning" the repository at https://github.com/skiadas/li
There are start files there for you to work with. You can clone this repository by going to an appropriate location in your terminal, and run the commands:

```
git clone https://github.com/skiadas/linky.git
cd linky
```

Here are the files included in this folder, named linky:

- A file that handles your database calls: db.py or some similar name.
- A file that handles the web service stuff: main.py or some similar name.
- A file that contains test calls: tests.py or something similar.

When you are ready to submit your assignment, you should create a zip file of the linky folder, and email it to me. To create a zip file you can go to the *parent* directory of the linky folder in the terminal, and run the command:

```
zip -r yourFavoriteName.zip linky
```

Database requirements

Your database file should follow the structure of the sample db.py, and use ORM to introduce needed classes, followed by a Db class that represents a connection to the database. A start on these has been done for you. Here are the classes you should have:

- Shortcut is a class representing a link's shortcut. It contains the following table fields:
  - linkHash is of String type, and must not be null. It is a primary key, together with the bucketId.
  - bucketId is of String type, and must also not be null. It is a primary key, together with the hash. It is also a foreign key pointing to the bucket id field. When a bucket is deleted, all the corresponding shortcuts that were stored in it must also be deleted, so make sure to set the ondelete setting correctly.
  - link is of String type, and must be not null.
  - description is of String type, it is allowed to be nullable.

  The Shortcut class also has a relationship bucket to the Bucket class, whose opposite via back_populates is shortcuts.
- Bucket is a class representing a bucket of links. It contains the following table fields:
  - id is of String type, must not be null, and is the primary key.
  - description is of String type, and it is allowed to be nullable.
  - passwordHash is of String type, and must be not null.

  TheBucket class also has a relationship shortcuts with the Shortcut class, whose opposite via back_populates is bucket.

Tasks, part 1: Add the suitable lines to db.py to set up the above classes. **Tasks, part 2**: Your Db class should provide at least the following methods (there are tests provided for these methods in the tests.py file, make sure you can pass those tests, and you should add your own as well):

1. A method getBuckets(self) that returns a list of all buckets in the system. The result should be a (possibly empty) list of Bucket objects.
2. A method addBucket(self, id, passwordHash, description=None) that can be used to create a new bucket in the database. The method should return the created bucket object. This method does NOT need to check if a bucket with that id already exists, that's the job of its caller.
3. A method getBucket(self, id) that is given a bucket id and searches in the database for a bucket with that id. Returns either the Bucket object with that id, or None.
4. A method deleteBucket(self, bucket) that can be used to delete an existing bucket. It expects to be given a bucket object, and deletes it from the database.
5. A method addShortcut(self, linkHash, bucket, link, description=None) which creates a new shortcut entry in the database. It expects to take a shortcut string, a Bucket object, a link string and optionally a description string. It does NOT need to check if an entry with the same primary keys exists, that's the job of its caller.
6. A method getShortcut(self, linkHash, bucket) that searches for a shortcut with a given hash string and belonging to a given bucket, and returns it if one exists (or returns None otherwise).
7. A method deleteShortcut(self, shortcut) that is given a Shortcut object and deletes it.

Web Service requirements

You will implement the following web service routes (details on them follow):

Route Scheme	Function Name	Role
GET '/'	bucket_list	List of all buckets
GET '/<bucketId>'	bucket_contents	Contents of a bucket
POST '/'	bucket_create	Create a new bucket
PUT '/<bucketId>'	bucket_create_with_id	Create a new bucket
DELETE '/<bucketId>'	bucket_delete	Delete a bucket
GET '/<bucketId>/<hash>'	shortcut_get_link	Get a shortcut link
POST '/<bucketId>'	shortcut_create	Create a shortcut link
PUT '/<bucketId>/<hash>'	shortcut_create_with_hash	Create a shortcut link
DELETE '/<bucketId>/<hash>'	shortcut_delete	Delete a shortcut link

Add tests for these in the tests.py file. The file has some initial entries created for you

Tasks, part 3: You must implement the following services. You may create any helper functions you like, similar to what we did in the banking–flask project.

1. GET '/' is meant to return a list of all the available buckets, and it is implemented in the function bucket_list. You should return a payload that looks as follows:

```
{
    "buckets": [
        {
            "link": "/<bucketID>",
            "description": "the bucket description"
        },
        ...
    ]
}
```

   You can use a list comprehension over the result of the database query to create that list of bucket entries. Use url_for to create the links.

2. GET '/<bucketId>' returns information about a particular bucket. It is implemented in the function bucket_contents. Its behavior should be as follows:

   - If the bucketId does not match a bucket in the database, return a 404 status code (use abort for this and other error behaviors, helper error handlers are provided at the top of the file).
   - Get the request's arguments and check if one of the argument's is a password field. If there isn't one, return a 403 status with a message that a password is required.
   - If there is a password field, use the getHash function from the utils library to convert this password value into a hash, and compare it to the stored passwordHash in the bucket. If they don't match, return a 403 status with a message about incorrect password.

- If the password does match, then you must return a normal response. Your payload should have the following structure:

```
{
    "id": "the bucket id",
    "link": "the link back to this bucket, use `url_for`",
    "description": "the bucket's description",
    "shortcuts": [
        {
            "linkHash": "the linkHash of the shortcut",
            "link": "link to the shortcut, use `url_for`",
            "description": "the shortcut description"
        },
        ... more shortcuts here
    ]
}
```

3. PUT '/<bucketId>' is used to create a new bucket with a given id provided by the user. It is implemented in the bucket_create_with_id function. Its behavior should be as follows:

    - If there is already a bucket with that bucketId, you should return a 403 status, stating that this bucket id already exists.
    - You will need to read the contents of the payload submitted to you, in JSON form. If those contents do not contain a "password" field, you should return a 403 status, stating that a password for the bucket must be provided.
    - If there is a password field, then you should use the getHash function from the utils library to convert it into a linkHash value. You should also look for a "description" field in the contents of the submitted payload. Then you should use the suitable db method to create a bucket with the provided id, linkHash, and description if one was provided, and ask for a commit from the db object. You should then return a 201 status, along with a Location header whose value should be the url_for link to this newly created bucket's GET method. Your body can be an empty JSON ({}) or one that has an "ok" entry.

4. POST '/' is used to create a new bucket with an automatically-generated id. It is implemented by the bucket_create function. It behaves similarly to the PUT version above, with a small variation:

    - Use the makeId function from the utils module to generate a random bucket id.
    - Check if a bucket with that id already exists, and if it does keep generating new id's until you find one that doesn't exist already.
    - Return the result of calling the bucket_create_with_id function, passing to it the generated id. This will ensure that the rest of the method's behavior follows that of the PUT version of creating a bucket.

5. DELETE '/<bucketId>' is used to delete a bucket. It is implemented in the bucket_delete function. Its behavior should be as follows:

- Search for a bucket with that id. If there isn't one, return a 404 status.
- If there is such a bucket, you should look at the parameters/arguments passed along with the DELETE request. If there is no password argument provided, return a 403 error, with a message that a password is required.
- If a password was provided, use the getHash function from the utils module to convert the password into a passwordHash. Then check whether this passwordHash matches the one stored in the retrieved bucket. If they do not match, return a 403 error, with a message about an incorrect password.
- If the passwordHash matches the one from the bucket, then you should ask the database to delete the bucket, then commit. You should then return a 204.

6. GET '/<bucketId>/<hash>' is a method used to retrieve a shortcut. It is implemented in the function shortcut_get_link. Its behavior should be as follows:

   - Search for a bucket with that id. If there isn't one, return a 404 status.
   - If there is a bucket with that id, then use a suitable db method to search for a shortcut with that linkHash and belonging to that that bucket. If one is not found, then return a 404 status.
   - If such a shortcut is found, then return a 307 response code, indicating a redirect. Set the Location header of your response to the link value from the shortcut. You must also include a payload with your response, containing:

     ```
     {
         "hash": "the linkHash",
         "link": "the shortcut target link",
         "description": "the shortcut description"
     }
     ```

7. PUT '/<bucketId>/<hash>' is used to create new shortcuts. It is implemented by the function shortcut_create_with_hash. Its behavior should be as follows:

   - Search for a bucket with that id. If there isn't one, return a 404 status, with a message mentioning that the bucket id is unknown.
   - Search for an existing shortcut with that link hash and that bucket. If one exists, return a 403, indicating that the hash is already used.
   - If such a shortcut doesn't exist already, then look at the contents of the request. If they don't contain a "password" field, then return a 403 status, with a message about a password being required.
   - If a password field exists, use getHash to convert it to a passwordHash and test it against the one stored with the bucket. If they don't match, return a 403 status, with a message about an incorrect password.
   - If the passwordHash matches the one for the bucket, then create a new shortcut entry using the appropriate db method, issue a commit, and return a 201 status code along with a Location header pointing to the url_for entry for a GET for this shortcut.

8. POST '/<bucketId>' can also be used to create new shortcuts, where the client expects the linkHash to be generated by the server. It is implemented by the function shortcut_create. Its behavior should be as follows:

    - Use makeId to generate a value for linkHash.
    - If there is already a shortcut for the provided bucketId and the generated linkHash, then keep generating new linkHashs until that's no longer the case.
    - Return a call to the shortcut_create_with_hash function, passing to it the provided bucketId and the generated linkHash.

9. DELETE '/<bucketId>/<linkHash>' is used to delete a shortcut. It is implemented by the shortcut_delete function. Its behavior should be as follows:

    - Check if the bucket with that bucketId exists. Return a 404 if it does not exist.
    - Check for an existing shortcut with that bucket and the provided linkHash. Return a 404 if it doesn't exist.
    - If the shortcut doesn exist, then look at the request's arguments, for a password entry. Return a 403 with a message that a password is required.
    - If a password entry is provided in the request's arguments, then use makeHash to convert it to a passwordHash and compare it to the bucket's passwordHash. If they don't match, return a 403 with a message that the password is incorrect.
    - If the passwordHash matches the value in the bucket, then use a suitable db method to delete the shortcut, make a commit, and issue a 204 response.