The System

You're tasked with creating a simple transactional HTTP key value database which responds to the following routes. Each of these routes, if it takes a payload, expects that payload as valid JSON.

- `GET /thing` should return the value of the key `thing` with status code `200`, or a `404` if the key does not exist, like so:

  ```
  $ curl -X GET http://localhost:4000/thing
  {"thing":"value"}
  ```

- `POST /set` should take a JSON payload which includes a `key:value` pair which should be queued for saving when the `/commit` route is hit, and returns a `201` *if the value is new*, like so, along with the `key:value` being saved:

  ```
  $ curl -X POST http://localhost:4000/set -d  '{"thing": "value"}' -H
  "Content-Type: application/json"
  {"thing":"value"}
  ```

  If the value will *not be new*, and is instead being updated, the server should return a `200`. A `POST` should never be able to update more than one key at a time, and should return a `400` if the user attempts it.

- `POST /commit` should take all the `key:value` pairs passed to the server via the `POST /set` route above, and save them to the database, and return a `204`

- `DELETE /set` should take a JSON payload containing a key to mark for deletion. This route acts just like `POST /set`, in that the key is not erased until the `POST /commit` route is hit

  ```
  $ curl -X DELETE http://localhost:4000/set -d '"thing"' -H "Content-
  Type: application/json"
  ```

  Unlike the `GET` route, this route should always return `200`. As in, you should be able to "erase" a key which does not exist and the system should not complain

Hitting routes which are not specified above or using methods on routes that are not specified above should result in a `404`.

To reiterate, all routes, if they take payloads, expect them as valid JSON. All responses are also expected to be valid JSON, and do not need to be formatted neatly.

For example, the server should return the following from this series of example `curls`, with simplified output:

```
$ curl -X GET http://localhost:4000/key
404
$ curl -X POST http://localhost:4000/set -d '{"key": "value"}' -H "Content-
Type: application/json"
201
$ curl -X GET http://localhost:4000/key # Note here the key has yet to be
_committed_
404
$ curl -X POST http://localhost:4000/commit
200
$ curl -X GET http://localhost:4000/key # The key has now been committed
{"key": "value"}
$ curl -X DELETE http://localhost:4000/set -d '"key"' -H "Content-Type:
application/json"
200
$ curl -X GET http://localhost:4000/key # The key still exists at this stage
{"key": "value"}
$ curl -X POST http://localhost:4000/commit
200
$ curl -X GET http://localhost:4000/key # The key no longer exists
404
```

Further Requirements

This system should persist reboots of the server itself.

This system can be written in any language, though we commonly recommend something like Python, Ruby, Go, or node.js.

You may use any third party libraries or dependencies you like, assuming you can package them with your project as described below.

Included within a tar.gz or .zip file of your entire project should be the following files:

`README.md`: A brief description of the inner workings of your code, plus a list of the third party libraries or other projects which you used to complete this assignment, along with a short description of why you chose to use that resource. If there are any improvements you would make to the system if you had more time, please list them here as well.

`build.sh`: A script which does all the necessary work to setup and/or build your project

`run.sh`: A script which runs the server on localhost port 4000

How we'll evaluate it

We have a testing framework which will build and run your system using the scripts mentioned above, then make a series of automated requests to your system to test its functionality. *However*, of highest importance to us is clean, well-organized code, not 100% test result perfection.

During your on-site interview we'll discuss your submission, and focus especially on your thoughts on the following questions:

1. Do you notice anything deficient about the API as defined above?
2. Are there any changes you would make to the API to improve it?
3. If this server is running on a system with constrained memory, what measures would you have to take to ensure it does not crash due to a lack of available memory?
4. How could you improve the performance of the system for frequently accessed keys?