CS 130 Hub

Assignment 7

In this assignment you will do part of the work in your own code base, and part in another team's codebase. By the transitive, reflective, or some other property(?), this also means some team will be working as a *contributor* in your codebase. The list of cross-team assignments can be found here.

Contributors will be able to clone repositories and create and update changes, but will not be able to approve or submit changes. It will be up to the repository-owning team to approve and submit changes.

In order to minimize cross-team review delays, we will allow any team member to review incoming changes *from the other team* for this assignment. It will still be up to the TL to drive the reviews and make sure someone on their team (or themselves) is on top of the reviews. If your team keeps the other team waiting too long (more than 6 business hours), you may be penalized in your grade. Aim for 2-4 hours turnaround time during the day. If things are running slow due to negligence of one team over the other, the negligent team will be the one who receives a late submission penalty, as opposed to a team who is very responsive and on top of things. Once you approve a change, you should immediately submit it.

In short, your team is responsible both for getting code approved in another team's codebase, as well as approving and submitting another team's code in your own codebase. A portion of your grade on this assignment will be based on the other team's ability to get part 1 working and submitted.

Each student should submit this assignment by 11:59PM on May 23, 2023 into the submission form.

TABLE OF CONTENTS

CRUD API handler

Written in the other team's code base

Many web applications reply on server-side data stores to save application data to later serve to users. The ability to Create, Read, Update, and Delete such information is commonly referred to as CRUD, and a simple CRUD API can be used to power even complex-looking web or mobile applications. In this assignment you will write a simple CRUD API request handler that allows the storage and retrieval of JSON (text) information for Entities, referenced by an ID.

CRUD basics

We will refer to types of objects that can be saved as *Entities*, and we will refer to unique identifiers for specific instances of an Entity as an *ID*. The basic lifespan of data through a CRUD API is as follows:

- An instance of Entity is **created** with some data, and an ID referring to the newly created instance is returned by the API.
- Entity data can be **retrieved** by calling the retrieval API with the ID referring to a previously created object.
- An instance of a specific Entity can be **updated** with new data by calling the update API with the ID of a previously created object.
  - This *may* also be used to create an Entity with a specific pre-determined ID.
- An instance of a specific Entity can be **deleted** from the system by calling the delete API with the ID of a valid object.

You can find a straightforward example of a CRUD API at CrudCrud. This is a valuable resource for playing around with the different operation types to build a mental model of how data is stored and represented. The API defined by CrudCrud is very similar to the API we define below for this assignment, but **not** exactly the same. Please take care to understand the requirements in the assignment and not to accidentally implement the API contract defined by CrudCrud instead.

API request handler

A new API request handler should implement a CRUD API as follows:

- Allow upload of JSON data without any ID with an HTTP `POST` of JSON data (in the `POST` body), returning the ID of the newly created object as JSON.
- Allow retrieval of JSON data for a given ID with an HTTP `GET` with the ID in the request URL.
- Allow upload of JSON data to a specific ID with an HTTP `PUT` of JSON data (in the PUT body).

- Allow deletion of stored data for a specific ID with an HTTP `DELETE` with the ID in the request URL.

The API request handler should listen on some prefix (e.g. `/api`), and support CRUD operations for arbitrary Entity types. For our purposes, an Entity is defined by simply a name (e.g. "Shoes") that can define independent ID spaces. (`GET /api/Shoes/1` and `GET /api/Books/1` can return separate objects, even though they both share ID 1)

In addition, the request handler should be able to list entities of a given type as follows:

- Allow retrieval of existing IDs within an Entity with an HTTP `GET` with the Entity type in the request URL (and no ID).

("List" doesn't fit nicely into the CRUD acronym, but is often important for real usage of these sorts of APIs)

You will need a persistent key-value store to act as the backend storage system / database. To keep it simple, we can use the server's filesystem to store and retrieve data by ID in directories. The API request handler should be configurable, with a `data_path` parameter specifying the root directory of created data.

Example

Consider the following examples for working with an Entity named **Shoes**, a `data_path` of `/mnt/crud`, and an API request handler listening at `/api`:

Action	Example request	Visible behavior	Internal behavior
Create	`POST /api/Shoes` (data in POST body)	Returns the ID (1) of a newly created shoe.	Finds next available ID (1) and writes POST body to `/mnt/crud/Shoes/1`. Returns `{"id": 1}` in the response.
Retrieve	`GET /api/Shoes/1`	Returns the data given in the previous step for shoe 1.	Reads `/mnt/crud/Shoes/1` and returns to user.
Update	`PUT /api/Shoes/1` (data in PUT body)	Updates the data for shoe 1 with new data.	Writes PUT body to `/mnt/crud/Shoes/1`.
Delete	`DELETE /api/Shoes/1`	Delete shoe 1 from the system	Removes file `/mnt/crud/Shoes/1`.

Action	Example request	Visible behavior	Internal behavior
List	`GET /api/Shoes`	Returns a JSON list of valid IDs for the given Entity.	Lists filenames in `/mnt/crud/Shoes`. Returns the list of file names, e.g. `[1]`, in the response.

Additional requirements

You should be able *unit test* your request handler using dependency injection. (Read: you will need a fake version of the filesystem or some other interface for file I/O).

You should write an *integration test* for the request handler. The integration test should exercise the following:

- Creating an entity
- Retrieving that entity and verifying its correctness
- Deleting that entity and verifying that the entity is no longer retrievable

You should get this request handler working in another team's codebase, by running their server locally.

Other considerations

Until now, we have worked with a few well-known HTTP response codes (e.g. 200 and 404). CRUD operations may have other response codes that are more appropriate for the given scenario.

We have included detailed examples of how this CRUD API should work in the basic cases. However, edge cases remain unspecified. Consider these edge cases, come up with reasonable behaviors, and document these behaviors thoroughly (both in code and in team notes).

Note: when thinking of these edge cases and potential behaviors, consider that the user of this request handler is not an end user (since a browser will not send `POST` or `DELETE` when navigating through the URL bar), but rather other developers. How would they expect your API to behave? What responses and/or behaviors would be useful to a developer when they hit edge cases?

Deploying CRUD

Optional: Done in your own project

Once the other team has finished submitting the CRUD API handler, you should get it working on your GCP deployment. In order for your data to persist across container and VM instance starts, you

will need to allocate a new Persistent Disk in Compute Engine, attach it to your VM, and mount it to your container to make it available to your server at the configured `data_path`.

Note: *This functionality will be required and graded in the following assignment.*

Grading Criteria

The minimum requirements for this assignment are drawn from the instructions above. The team grading criteria includes:

- Implement and test a CRUD API handler in another team's server
- Responsiveness to the team working on your server (as rated by the other team's TL)
- Good documentation for the other team (as rated by the other team's TL)

Individual Contributor criteria includes:

- Code submitted for review (follows existing style, readable, testable)
- Addressed and resolved all comments from TL

Tech Lead criteria includes:

- Kept assignment tracker complete and up to date
- Maintained comprehensive meeting notes
- Gave multiple thoughtful (more than just an LGTM) reviews in Gerrit

General criteria includes how well your team follows the class project protocol. Additional criteria may be considered at the discretion of graders or instructors.

Submit your assignment

Everyone should fill out the submission form before 11:59PM on the due date. We will only review code that was submitted to the `main` branch of the team repository before the time of the TL's submission. You may update your submission by re-submitting the form, which will update the submission time considered for grading purposes if you are the TL.

Late hours will acrue based on the submission time, so TL's should avoid re-submitting the form after the due date unless they want to use late hours.

"You may think using Google's great, but I still think it's terrible." —Larry Page