17625 – API Design
Assignment 1.2: RESTful API
Shashank Shekhar, sshekha3

Reflection:

1. **What were some of the alternative design options considered? Why did you choose the selected option?**
   Answer: I have created the given API using REST principles. Alternative to RESTful API included creating a SOAP architecture or GraphQL. The API had different requirements and catered to a diverse audience, as such REST API best suited the needs because:
   a. It is lightweight. This API will potentially be used by a large customer base and as such it needed to be lightweight to be adjusted and modified.
   b. It is also independent of the client implementation therefore REST helped create client agnostic API which can be used by diverse users.
   c. It is also highly scalable and easy to modify. Individual endpoints and services can be modified according to the need and therefore scaling is easy.
   d. It is easy to document. The RESTful approach adds an extra level of abstraction that makes it easy to convey technical details through simple endpoints which can be consumed without worrying about the actual implementation or needing to write any query as in GraphQL.

   Also, while designing the API, different endpoints were required. For instance, an endpoint to get current day, another one to get current month, etc. One design patter would have been to implement all these endpoints separately as follows:

Requirement	**Possible endpoint**
Get Current day	GET/currentDay
Get Current month	GET/currentMonth

   However, since these end points all revolve around date functionality, they were clubbed together as follows:

Requirement	**Actual endpoint**
Get Current day	GET/date/currentDay
Get Current month	GET/date/currentMonth

   This design patter not only makes the development easy, since endpoint with “date” handle date related functionalities, but also made consumption easy. Client can look for “date” endpoints in the documentation to find an endpoint that suits their need. Similar clubbing was done for “events” endpoint where we were fetching all events related data as shown below:

Requirement	**Actual endpoint**
Get events for a given date	GET/events/date

Get event by id	GET/events/:eventId

Above image makes it convenient for the user to find “event” related endpoints at one place.
Similarly, another design choice was to provide time zones to the clients. According to the problem statement, the API will be used internationally therefore clients might reside at different places. As such, it is necessary to serve data suitable to the client needs. This is clearly represented in the documentation as well and helps in wider adoption of the API.

2. **What changes did you need to make to your tests (if any) to get them to pass. Why were those changes needed, and do they shed any light on your design?**
Answer:
The test cases written earlier had several shortcomings which are as follows:
   a. The mock data used earlier was from a JSON file which made its consumption difficult.
   b. The tests did not consider how the server will be run for individual test suites.
   c. The servers did not correctly equate the response body and response body has changed during the implementation.
   d. Response codes did not match since according to the condition, response codes have changed.

   To solve these issues, firstly a separate test server is being run for individual test suites. This removes any conflict in the test suites and ensures that all test cases run smoothly. Second, the data is mocked separately for individual tests rather than use similar data as the data is being modified during each test. Thirdly, status codes have been modified accordingly to check for each type of response. For instance, 204 has been introduced when a content is not found. 400 is used to signify that the data sent by the user is incorrect.
   On similar lines responses have also changed to better illustrate the root cause. For instance, if the id of an event is passed incorrectly, user is notified of it through response.

   These changes were necessary because they made testing the API easier. Also, these changes covered a breadth pf edge cases and scenarios making API capable of handling edge cases smoothly. This ensures that when API does rollout, it is easily adopted by the audience without causing unwanted behavior that might dampen its adoption.

   The changes implemented imply that the initial design was insufficient and did not consider edge cases. It was also lacking in terms of the various responses that would need to be sent to the client to make it easier to use the API. For example, if a user sends incorrect time zone and is never notified that the time zone is wrong, they might not be able to figure out the same and use the API as intended. Therefore, it is necessary to test all these cases and ensure that user receives the right information.

3. **Pick one design principle discussed in class and describe how your design adheres to this principle.**

   Answer:

   The API is easy to use and easy to learn. It is easy to learn because the endpoints have been defined in such a way that for a given segment of functionality, user can easily search for their needs and access the information they need. For instance, the names have been declared so that they can be used intuitively. For example, if a user wants to search for current day, they need to access the endpoint "GET/date/currentDay".

   Similarly, if thought of intuitively to search for current month, they just need to replace "currentDay" with "currentMonth" and they will be served with the right information. Also, they have an option to fetch the supported date formats for the API so that they can refer to that to know which formats are supported and how their application should adhere to that.

   All these features not only make the API easy to learn but also easy to use. All the endpoints have been clearly defined to what parameters they take and expected output for the same. The consistency in the output enforced through schema validation makes sure that every time a uniform data is served. The use of appropriate error responses also keeps the client informed on what are the expectations of the API and how to modify their request to observe desired behavior. This also makes the API hard to misuse as the endpoints that are not defined simply inform the user that the resource does not exist. Also, since the endpoints validate data, any data outside of the bound of accepted data simply inform the user of incorrect data.

   The API being implemented is for people who want to use the services to build on top of that and might not necessarily include developers who have the knowledge to understand the technical detail. It is therefore beneficial that the technical detail is abstracted away by exposing easy to use endpoints.