Assignment A1.2: RESTful API

Reflection

1. **What were some of the alternative design options you considered? Why did you choose the selected option?**

   Below are some of the alternatives that I considered during the design process:
   - deleteAppointmentById()
     Implementing the method to delete an appointment using the POST method rather than the DELETE method. However, I proceeded with DELETE since it is idempotent in nature, i.e., multiple deletion requests won't be processed in case of duplicate requests. Our system was suitable to support the DELETE method as the deletion takes place on the basis of unique IDs assigned to individual appointments.

   - updateAppointment()
     Implementing the method to update/modify an appointment with the help of the POST method in place of the PUT method. But I utilized the PUT method for the same reason started above as PUT performs operations in an idempotent fashion as well and duplicate requests are not processed. Therefore, the idea to use the POST method was discarded for this specific scenario.

   The methods mentioned above are demonstrated in the image below.

```
// delete method to delete an appointment by ID
@DeleteMapping(path = "{id}")
public void deleteAppointmentById(@PathVariable("id") UUID id) { appointmentService.deleteAppointment(id); }

// put method to update an existing appointment by id
@PutMapping(path = "{id}")
public void updateAppointment(@PathVariable("id") UUID id, @Valid @NotNull @RequestBody Appointment appointmentToUpdate) {
    appointmentService.updateAppointment(id,appointmentToUpdate);
}
```

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?**

   Some of the changes I had to make to my previous test cases were:
   - Updation of input parameters
     Following the implementation of the working code, I added some constraints to the input, such as the "name" of the patient cannot be an empty string when adding a new appointment. A few adjustments were required to modify the existing test cases as a result of these changes. It also had an effect on the currently defined "happy" and "failing" cases, which were updated as well.

- Restructuring of endpoints
  Initially, I had designed multiple APIs with different endpoints to differentiate between the different operations. As a result, redundant and unnecessary test cases were generated. Grouping APIs with similar functionalities under the same endpoints and introducing parameters significantly simplified the code and test cases. Furthermore, it added extensibility and scalability to the existing code structure.

These modifications brought the system closer to a real-world application by identifying and resolving some major loopholes. It also made the code more readable and modular from the standpoint of a developer.

Making these changes made me realize how important it is to understand the system's requirements and applications when designing test cases. Taking a backward approach to designing this set of APIs, first dealing with test cases and then defining performance, allowed me to gain a better understanding of identifying system boundaries, handling edge cases, and the scope of the project at hand.

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

   One design principle that is illustrated in my design is **'Easy to read and maintain code that uses it'**. Some of the examples are:
   - Methods are concise and redundancy in the lines of code is minimal.
   - The naming conventions used for methods as well as variables are self-explanatory, making it easier to refer to the correct part of code.
   - Comments are added to give a brief overview of each part of the code.
   - Cohesion has been leveraged and maximized by structuring the code in a way that common files are grouped together under folders with simple naming conventions. Additionally, similar methods are grouped together under common classes and are contained in the same files.
   - Code modularity and reusability are supported due to this kind of file structuring.
   - Coupling is minimized by limiting the functionalities of individual modules.