Report

General Approach

The development operations stages can be separated into these main phases, listed below with their main objectives:

1. **Plan**: In this stage, the objective is to define the project goals, set requirements and select the tools and technologies that will be used. Writing the OpenAPI definition using SwaggerHub in this stage helps in defining the project goals and requirements.

2. **Create**: In this stage, the objective is to develop the code that will be used for the application. Generating the service stubs in Python using the OpenAPI definition from SwaggerHub in this stage helps to create a template of the service code.

3. **Verify**: In this stage, the objective is to ensure that the code works as expected and meets the project requirements. Implementing the logic in this stage helps to create the functionality of the service.

4. **Package**: In this stage, the objective is to package the application and its dependencies into an image that can be easily deployed. Building and testing the Docker image in this stage helps to ensure that the image is properly configured and can be used to deploy the service.

5. **Test**: In this stage, the objective is to test the application and ensure that it works correctly. Writing tests in this stage helps to ensure that the application is working as expected and that new changes do not introduce bugs or regressions.

6. **Release**: In this stage, the objective is to deploy the application to a production environment. Deploying the web service on Kubernetes (MicroK8s) in this stage helps to ensure that the application is available to users and that it is running in a stable and scalable environment.

All in all, these six stages help to ensure that the application is properly planned out, developed, tested, packaged, and deployed in a controlled and automated manner. The purpose of these stages is to reduce errors, ensure stability, and increase the speed of delivery.

OpenAPI: Define Objects

OpenAPI is a popular tool used for defining RESTful APIs, including defining the structure of the data that is being transmitted. Defining objects in OpenAPI allows developers to clearly define the structure of the data that their API is working with.

In this specific example, we are instructed to define two objects, Student and GradeRecord, using the OpenAPI specification. The Student object has four properties, including student_id, first_name, last_name, and gradeRecords. The gradeRecords property is an array, which can hold multiple GradeRecord objects. The GradeRecord object has two properties, consisting of subject_name and grade.

The student_id property could be defined with the format int32 since the int32 format specifies that the value of the property should be of type 32-bit integer, having a maximum value of 2,147,483,647. The use of int32 format for the student_id property will depend on the specific requirements and constraints of the system being developed. Using a larger data type such as int64

would take up more storage space, which could impact system performance and scalability. In general, the specific data type and format used for a student ID will depend on the requirements of the system being developed, but using an integer data type such as int32 is a common and practical choice.

To define these objects in OpenAPI, we need to create a new node called components, and within that node, create another node called schemas. Inside the schemas node, we define our objects. The resulting code is visible in Swagger Hub at the end of the OpenAPI file.

In the code above, we first define the Student object, including its required properties and the array of GradeRecord objects, as well as defining the data types for each property. Next, we define the GradeRecord object, including its required properties and the data types for each property. The grade property is defined as a number, with a minimum value of 0 and a maximum value of 10.

To conclude, defining objects in OpenAPI allows for clear and concise documentation of the structure of the data that is being transmitted. By defining the objects in this way, developers can more easily understand the structure of the data and ensure that their API is properly handling and transmitting this data.

OpenAPI: Delete Method

n OpenAPI, the HTTP DELETE method is typically used to delete a resource on the server. The OpenAPI specification provides a way to document this method as part of your API's interface.

To define a DELETE method using OpenAPI, you can follow these steps:

1. Add a new path object in your OpenAPI document to specify the URL of the resource you want to delete, using the `delete` operation.

2. Define any parameters required for the DELETE operation. For example, you may want to specify a path parameter to identify the specific resource to be deleted.

3. Define any request body required for the DELETE operation, if applicable. In some cases, the DELETE operation may not require a request body.

4. Define the response that will be returned when the resource is successfully deleted.

Here is an example of how you can define a DELETE method for an API endpoint that deletes a user with a specific ID:

```
yaml
paths:
  /student/{student_id}:
    delete:
      summary: deletes a student by ID
      description: Deletes a single student with the specified ID
      parameters:
        - name: student_id
          in: path
          description: UID of the student to delete
          required: true
          schema:
            type: number
           format: integer
      responses:
        '200':
          description: OK, successful operation
        '400':
          description: Bad request, invalid ID
        '404':
          description: ID not found
```

In this example, we define a path for deleting a user with the specified ID, using the DELETE method. The path includes a path parameter `userId` to identify the user to be deleted. We also define two possible responses: a successful `204` response with no content, and a `404` response to handle the case where the specified user is not found.

Note that the specific details of your DELETE method will depend on your API's requirements and the specific resource being deleted. However, this example should provide a good starting point for defining a DELETE method using OpenAPI.

Alpine Linux Advantages and Disadvantages

Alpine Linux is a lightweight and security-focused Linux distribution that is often used in Docker containers due to its small image size and fast boot time. Here are some advantages and disadvantages of using Alpine Linux in Docker:

Advantages:

1. Small image size: Alpine Linux has a very small image size, making it ideal for use in Docker containers where image size and network transfer time are important considerations.
2. Fast boot time: Alpine Linux has a fast boot time due to its small size, which is useful for applications that need to start quickly and respond to requests promptly.
3. Security-focused: Alpine Linux has a strong focus on security, with a minimal package set that reduces the attack surface of the container.
4. Package management: Alpine Linux uses the *apk* package manager, which is lightweight and fast, making it easy to install and manage packages in a container.

Disadvantages:

1. Limited package availability: Due to its minimal package set, some packages may not be available in Alpine Linux, which can make it challenging to build certain applications in the container.
2. Different package versions: Packages in Alpine Linux may be of different versions than those in other Linux distributions, which can cause compatibility issues with certain applications and libraries.
3. Limited documentation: Due to its smaller user base, Alpine Linux may have limited documentation compared to other Linux distributions, making it more challenging to troubleshoot issues.
4. Learning curve: Due to its minimalist approach, using Alpine Linux may require a steeper learning curve for those who are not familiar with its package management and configuration options.

In summary, Alpine Linux is a good choice for Docker containers that require a small image size, fast boot time, and security-focused environment. However, it may not be suitable for all applications, particularly those that require specific package versions or extensive documentation.