RESTful API Design: A Device Deployment Microservice

CS4991 Capstone Report, 2022
Boheng Mu
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
bm5dj@virginia.edu

Abstract

A Seattle-based cloud service company provides a deployment and development platform for IoT devices. The testing framework for their service is monolithic and is deployed per test case, resulting in inefficient allocation of resources and interference from test cases using the same devices. As an intern, I was tasked with decoupling the device provisioning process as a micro-service in the form of a RESTful API designed to follow industry standards and internal specifications for path, method, and payload. The implementation exclusively utilizes company-provided technology and draws on industry practices such as testing and OOP. The API I designed was ultimately able to deploy devices for the company's cloud computing service and fully utilize the scaling capability that was ostracized before. I successfully demonstrated the API using an internal device testing service. However, due to the time constraint and the scope of the project, I had to forgo integration with the existing test framework. Further development is currently in the process for integrating with the exiting test framework and the physical device lab the team manages.

1. Introduction

Software testing is a very important part of the software development cycle. There have been many careers and companies that emerged for the purpose of more effective testing [1]. The industry has adopted continuous integration, continuous deployment/ delivery (CICD) as the norm for delivering agile products and maintaining software; and software testing is an important part of this CICD workflow [2]. However, finding the right balance for testing is a task that requires careful articulation. Lack of testing may result in faulty delivery of the product, while too much could interrupt the sense of continuous development, forcing developers to wait a prolonged time for test results, before moving onto the next step.

What is worse than the previous two cases, however, is faulty tests. Faulty tests prevent engineers from properly accessing the results of their product, requiring additional engineering hours on debugging the test rather than working the product itself. Ensuring the integrity and smooth running of tests, then, has become the focus of many companies, including the one where I did my internship.

2. Background

Orchestrator and DUT

The company has its own internal tools for the CICD workflow, including automated testing. However, due to the product being relatively new and the need to integrate with special embedded hardware, the team had to build the testing solution from the ground up. The result is a testing orchestrator where engineers who wish to test their features only have to worry about writing tests and which devices they want to test on. The orchestrator will connect to the device under test (DUT) and perform the test in place of an engineer. The orchestrator is also interfaced with the company's existing automated testing system where code changes to the product will be periodically tested. DUTs are also set up in both cloud computing units and physical device lab, in different regions. DUTs are organized into device pools, categorized by device type, operating system, and capabilities.

3. Related Works

For Representational state transfer (REST) and RESTful APIs, the industry has used Swagger.io [3] as a standard. According to Casas,

et. al, (2021), Swagger.io provides a contract for path definition, methods, and even testing for API design [4]. It is a structure that is not specific to any language or technology and allows many different services to talk with each other. This one key feature is critical because the product under test has been rewritten from the Go language to Java. The current orchestrator is written in Go while a new open-source orchestrator is under development in Java.

Compatibility between the two orchestrators is critical. One such use of Swagger API definition to bridge the gap between technology is in jQuery AJAX validation, as explained by Tashtoush, et. al (2019) [5]. The validation tool allowed different services in different languages to talk to each other thanks to the API specification where the sender knows what the receiver is expecting, and the receiver knows how to interpret the information that it was sent. I aim to achieve this type of consistence and compatibility with the device deployment API.

4. Project Design

The design of the project strictly adheres to the company's guideline for designing a project. Which starts from understanding the problem and customer needs. Then move fast, build iteratively, and break little.

4.1 Test Cases

The test cases are identified by tags where engineers can select a combination of tags on their devices. There is a predefined suite of tests as well as the option for engineers to bring their own test. Each test case run will instantiate a new instance of the orchestrator. However, the problem comes from each test case having its own orchestrator, meaning that no one test will have context on the overall tests being run. Additionally, a race condition can happen when multiple test case grab the same DUT and corrupt the testing process. By decoupling the device provisioning process as a stand-alone micro-service, I seek to provide the overall context needed to prevent the race condition. Additionally, optimization can be done for the queueing and ramp up of devices for testing

4.2 Onboarding Project

To help the interns to familiarize themselves with the product, the team has the interns build an onboarding project with the project. For my onboarding project, I built a sitting posture improvement system with IOT devices and the company's IoT software service. The systems have pressure sensors and cameras that can help detect improper sitting posture in the user and will notify the user via SMS or email to remind them to fix their posture. By completing the onboarding project, I was able to understand the purpose of the product, what kind of tests can be run on the device, and more importantly how the interaction among the software and the physical devices work.

4.3 Gathering Requirements

To start any project, the company has the policy to always start from the customer perspective, which necessitates gathering requirements from the user. During the early phases of the project, I set up a series of meetings with engineers on the team. Their roles range from builders of the product that needs to run test to testing engineers that built the orchestrator from the ground up. Additionally, I took this time to familiarize myself with the existing code base for the orchestrator to understand how the API will be used. Finally, the following requirements were derived for the API:

- As a tester who wants to grab a device, I need to be able to make a request for a DUT
- As a tester who wants to test, I need to have the connectivity information to connect the DUT
- As a tester I need to be able to check on the status of my request and cancel if needed.
- As a testing engineer, I need to have a holistic view of device usage and availability

4.4 Proof of Concept

Throughout the requirement process, I experimented with different concepts and ultimately landed on the reservation concept that best fits the use case. The reservation concepts describe this API as a "library" that vends devices, which users must return. Physical devices need to be reused, while virtual devices are recycled internally. The user may reserve one or more devices; the "library" then sets aside the

requested devices for a certain amount of time. The user may opt to return the device early, or the API will forcibly collect them by cutting off the connection to the device and changing the logins. This concept was re-created on the company's internal API system and was approved by the manager and mentor of the project.

4.5 Key Components

The final design of the API contains three main paths:

- root/{devicePoolName}
- root/{devicePoolName}/reservations
- root/{devicePoolName}/reservations/{reservationID}

User can call the GET HTTP method on the root/{devicePoolName} path to get general information about the device pool, such information will reside in the HTTP response body

User can call the GET method on the "root/{devicePoolName}/reservations" path to obtain information on the overall reservation status, such as how may devices are in use and usage history. This feature is limited to key users. The user can call the POST method on the same path to create a reservation. Inside the POST request the user can parameters such as how many devices they want and how long they want to keep the reservation. A reservation ID is returned to the POST method in the response body, this ID will be the only way the user can interact with the reservation they created.

User can call the GET method on "root/{devicePoolName}/reservations/{reservationID}" to obtain status on the reservation, whether its approved or not. If the reservation is approved, the connectivity information to the device will also be returned, such as IP address of the device and the SSH key pair required for connection. PATCH method can be called on the same path to update the reservation time needed. Finally, the DELETE method can be called on the reservation to indicate completion of testing, in which the devices will be recycled. This delete method will also be automatically called at the end of the reservation.

4.6 Creating a Reservation

In order to create a reservation, the POST method on the reservation path invokes a piece of code that creates an instance in the reservation database. The database is organized in their respective device pools. On the creation of an instance, the device pool is scanned for available devices. If there are enough available devices to match the reservation, the reserved devices will be marked as unavailable, and the reservation will be marked as ready. In the event that there are not enough devices, the reservation will be marked as pending, more devices will be prepared in the background, and the can routine will be called periodically.

4.7 Challenges

Due to the time constraints of the internship and the complexity of interfacing with physical devices, only virtual devices were considered for this API. Furthermore, since this is the first time I have worked with the technologies provided and the first time designing an API, much guidance was needed to complete this project. Finally, I have full ownership of this project, meaning that I had to build everything from the underlying infrastructure to the user interface.

5. Results

I created a working version of the API using the companies' internal technologies and hosted on a virtual private cloud. Internal engineers were able to interface to it and make appropriate API calls to obtain a device. This was demonstrated by a wrapper code that calls the API, obtains the connectivity information, then performs the test via the internal device tester. The API was able to vend out device(s) on different virtual device pools. These device pools are hosted on my personal testing account, however due to security reasons, I was not able to directly return the SSH keypair to the user. To amend this, I assumed that the user would have access to the key pair via a special indicator. Furthermore, the API was able to utilize the scaling technology in the virtual devices, which was ostracized in previous orchestrators. As reservation requests trickle in, the API can increase or decrease the number of devices on hot standby. This balances the cost of maintaining devices on standby and the cost of engineering hours wasted on waiting for a device.

6. Conclusion

This device deployment microservice that I created as a RESTful service allows the user acceptance test to be hassle free and more efficient. This enables engineers on the team to focus on building the product, rather than configuring and provisioning devices, resulting in better use of engineering hours. Furthermore, the service has a more holistic view on device usage and can more efficiently allocate resources based on demand. This helps the company to significantly reduce wasted resources in both devices and engineering hours. By isolating the duty of device deployment for testing in a separate service, the device contingency issue is solved with a central arbitrator that assigns devices on demand. The functionality of the service meets the basic requirements specified at the beginning of the project; however, there is more work to be done.

7. Future Work

Due to the time constraint and the scope of the project, integration with the current testing orchestrator was foregone and the API can only manage virtual devices provided by the company. The team manages a device lab with physical devices which have characteristics beneficial to testing, but managing its connection with the testing orchestrator is complex and calls for a separate project. Furthermore, the contingency issue is theoretically solved, but parts of the testing orchestrator must be rewritten to integrate with the device deployment API. As I returned for a second internship with the team, these two works are under way and I was involved in the design process of the second phase of the project.

References

[1] Arachchi, S. and Perera, I. 2018. Continuous integration and continuous delivery pipeline automation for Agile Software Project Management. 2018 Moratuwa Engineering Research Conference (MERCon) (2018). DOI:http://dx.doi.org/10.1109/mercon.2018.8421965

[2] Bureau of Labor Statistics, U.S. Department of Labor, *Occupational Outlook Handbook*, Software Developers, Quality Assurance Analysts, and Testers. (2022). Retrieved September 22, 2022 from https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm#:~:text=in%20May%202021.-,Job%20Outlook,the%20average%20for%20all%20occupations.

[3] Swagger. n.d. API development for everyone. Retrieved September 23, 2022 from https://swagger.io/

[4] Casas, S., Cruz, D., Vidal, G., and Constanzo, M. 2021. Uses and applications of the openapi/swagger specification: A systematic mapping of the literature. *2021 40th International Conference of the Chilean Computer Science Society (SCCC)* (2021). DOI:http://dx.doi.org/10.1109/sccc54552.2021.9650408

[5] Tashtoush, Y., Nour, M., Salameh, A. O., and Alsmirat, M. 2019. Swagger-based jQuery Ajax Validation. *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)* (2019). DOI:http://dx.doi.org/10.1109/ccwc.2019.8666542