An Advanced Resource Retention System for Integration Testing

A Capstone Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Nathan Snyder

May 12, 2023

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Nathan Snyder

Capstone advisor: Rosanne Vrugtman, Department of Computer Science

An Advanced Resource Retention System for Integration Testing

CS4991 Capstone Report, 2022

Nathan Snyder
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
nbs9vy@virginia.edu

Abstract

Valkyrie, a team within Amazon Web Services' Aurora service, found that some resources created during integration testing were not getting deleted, costing the company each second that they continued to run. The monetary cost of integration testing became unreasonably high due to all the resources that were not being used but had not been deleted. I designed and deployed an advanced resource retention system that tracks every resource used in integration testing and deletes them when appropriate. I created a Resource Metadata Table (RMT) to store metadata about each integration testing resource. I then made a sweeper tool that goes through the entries in the RMT and makes changes, such as deleting resources or removing them from the table. I also made a command-line tool for manually adjusting the expiration times of specific resources. After implementing all parts of the advanced resource retention system, all the unused resources that had been costing the company money had been deleted and the costs of performing integration tests decreased significantly. In the future, the system can be generalized for use by other teams within Amazon Web Services.

1. Introduction

In the previous integration testing paradigm, if a test passed successfully, the resources were deleted automatically. Should the test fail, or should an error be thrown during the test, the resources kept running indefinitely. The rationale behind this mechanism was rather simple: engineers need to be able to review what went wrong with the test. Only engineers could manually clean up the resources. However, this quickly became a nuisance, since in practice, most running resources from failed tests did not get cleaned up. Engineers who had other, more pressing work to do did not want to take the time to delete all the resources that their integration tests created. Running resources costs money and causes the clusters the resources are stored on to become bloated. Having these resources running posed a serious financial strain on the team due to the cost of storage space on the cloud. Instead of relying on manual clean-up, it was cheaper to use an automated method of cleaning up unused, still-running resources. Doing so induced a significant reduction in the financial losses incurred by the team as well as in the work of the engineers themselves.

2. Related Works

Efficiency is highly desired throughout all parts of computer science, and integration testing is no exception. In most cases, the optimization of integration testing is done through modifying the number or order of tests. There has been great research into this kind of integration test optimization, such as optimization by using automated UML diagrams and graph algorithms (Le Hahn, et al, 2001). Outside of integration testing, there has also been considerable research into automating the management of resources on

the cloud, specifically on Amazon Web Services. The goal of automated resource management is to oversee maintaining, monitoring, and cleansing large numbers of resources (Sheikh, et al, 2019). There has been little research into managing resources specifically for integration testing. In this project, I used ideas from automated resource management and brought them into the context of Valkyrie's preexisting integration testing system.

3. Project Design

The design of the resource retention system was developed over several weeks with input from my manager, my mentor, and the greater Valkyrie team. I gathered the requirements and considerations for the design, and, in the end, I developed the system by creating three components: the metadata creation and storage library (section 3.3), the sweeper tool (section 3.4), and the tool for adjusting expiration times (section 3.5).

3.1 Design Considerations

Valkyrie's resource retention system must keep track of every resource in the system and ensure that the resources get deleted when they are no longer useful. The whole process must run without human intervention, because relying on engineers to delete resources resulted in the resources not getting deleted. It was also necessary that the system be able to adjust in response to any special needs for specific resources. For example, if an engineer wants to ensure that a resource stay running for at least two weeks, the system should adjust accordingly and not delete it any earlier. Also, the system needs to be flexible and be able to accommodate every type of resource that can be used in integration testing.

3.2 Resource Lifecycle

Valkyrie's resource retention system must track resources from before they are created to after they are deleted. I created a lifecycle that models the different stages of every resource: *NOT_CREATED*, *CREATING*, *CREATED*, *DELETING*, and *DELETED*. All resources start with a status of *NOT_CREATED*, and with each run of the Resource Sweeper Tool (section 3.4), the status is advanced through the other stages until the resource is removed from the Resource Metadata Table.

Every resource has an expiration time for its status. When that expiration time is reached, the resource is moved to another status depending on the state of the resource and a new expiration time is given. The algorithm for determining the new status of a resource is discussed in section 3.4.

3.3. Metadata Creation and Storage Library

The resource metadata library is a Java library for creating metadata for a resource and storing it in the Resource Metadata Table. In the RMT, every row corresponds to a different resource. The columns are resource id, creation time, expiration time, location, owner, resource name, resource object, resource class, status, test id, and tags.

Each type of resource is modeled as a Java class that implements the *ValkyrieResource* interface. The *ValkyrieResource* interface specifies certain methods that every resource class must implement, including *close()* and *doesResourceExist()* methods. Resource classes implement those methods as well as any additional fields or methods that are needed to properly model that type of resource. To represent a specific resource, an object of the corresponding resource type class is created.

Every resource created for an integration test is added to the RMT by instantiating an object of the *ResourceMetadata* class. The constructor for this class takes in a resource object, formats the resource's metadata, and inserts it into the RMT. The *ResourceMetadata* class has additional methods for updating and saving the metadata

as well as for increasing, setting to an arbitrary time, and resetting to zero a resource's expiration time.

The *ResourceMetadata* class contains a static subclass called *ResourceMetadataUtils*, which contains several methods that retrieve or change a resource's metadata in the RMT given only the resource's id. For example, there are methods to get the resource's entire entry from the RMT; increase, set, and reset the resource's expiration time; delete the resource; and change an arbitrary attribute of the resource.

3.4. Resource Sweeper Tool

The Resource Sweeper Tool enforces the resource lifecycle on the resources in the Resource Metadata Table. It makes several queries to the RMT and adjusts the metadata depending on the state of the resources. I scheduled a job using Amazon Web Services' Distributed Job Scheduler (DJS) to automatically run the sweeper tool script once a day.

All resources are added to the table with a creation time of -1 and an expiration time of one week after the time the resource is added. When the resource reaches the *CREATED* status, the expiration time is reset to one week after the current time or -1 if the engineer who made the resource specified that this resource should not be deleted. When the resource reaches the *DELETED* status, the expiration time is also reset to one week after the current time.

On every run of the resource sweeper tool, the following changes are made for each resource:

1. If the resource's status is *NOT_CREATED* and it has passed its expiration time, the resource's metadata is removed from the RMT.
2. If the resource's status is *CREATING* and it has passed its expiration time, the existence of the resource is checked. If the resource exists, it is deleted. If the resource does not exist, the resource's metadata is removed from the RMT.
3. If the resource's status is *CREATED* and it has passed its expiration time, the resource is deleted.
4. If the resource's status is *DELETING* and the resource does not exist, the status is changed to *DELETED*. If the resource does exist, the "deletion attempts" column of the RMT is checked. If the number of deletions is less than three, the resource is deleted again. If the number of deletion attempts is greater than or equal to three, a ticket is created so that a human will manually delete the resource.
5. If a resource's status is *DELETED* and it has passed its expiration time, the resource's metadata is removed from the table.

3.5. Change Resource Expiration Time Tool

The Change Resource Expiration Tool script takes in several command-line arguments that identify resources and specify how to change those resources' expiration times. The script queries the Resource Metadata Table for the resources matching the given command-line arguments and modifies the expiration times for each one according to the inputs to the script.

The user can specify the resources they want to change by using the following flags: --resource-id, --resource-name, --resource-class, --test-id, --tag, --location, and --owner.

The --change flag tells the script how the user wants to change the resources' expiration times. The options are "increase," "set" (to an arbitrary time), and "reset" (set to time 0 so that the resource gets deleted on the next run of the sweeper tool).

If the user chooses "increase" or "set", time arguments should be specified using the flags --weeks, --days, --hours, --minutes, and --seconds. If no time arguments are specified, the time will default to 0. If the user chooses

"reset," any non-zero time arguments will cause an error to be thrown. The maximum the expiration time can be increased or set is two weeks after the current time. Any time later than that causes an error to be thrown.

4. Results

Currently, the advanced resource retention system I designed is in use by the Valkyrie team. The software moves all the resources created for integration tests through the life cycle and ensures that all resources get properly disposed of without the need for human intervention. This has resulted in considerable savings due to the problem first presented in the introduction, where running resources caused serious financial strain on the team. These savings are estimated to be in the thousands of dollars per year.

5. Conclusions

During my internship at Amazon Web Services, I designed an advanced resource retention system for the Valkyrie team. My project was to build a system to efficiently and automatedly delete integration testing resources after they are no longer needed. In building it, I designed a metadata creation and storage library, a sweeper tool, and a tool for adjusting resources' expiration times. The system is now currently in use by the Valkyrie team, where is has resulted in significant monetary savings.

6. Future Work

This advanced resource retention system was designed for the Valkyrie team, but most other software development teams would likely benefit from a similar system. The next step for this project is to generalize it for use by any team in Amazon Web Services. Also, there are many opportunities available for increased efficiency in integration testing apart from this system. One other avenue for optimization is to use different models of the systems that are being tested that optimize the number and order of the integration tests being run.

7. Acknowledgements

I would like to thank my manager, Bassu Hiremath, my mentor, Kevin Liu, and the entire Valkyrie team for their constant and supportive help with this project during my internship.

References

Le Hanh, V., Akif, K., Le Traon, Y., and Jézéque, J.M. 2001. "Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies," Lecture Notes in Computer Science, vol 2072, June 2001. doi:10.1007/3-540-45337-7_20

Sheikh, S., Suganya, G., and Premalatha, M. 2019. "Automated Resource Management on AWS Cloud Platform," Smart Innovation, Systems and Technologies, vol 164, October 2019. doi:10.1007/978-981-32-9889-7_11