CSTS

Enhancement Report

Esra Kastrati - 215507205

Mostafa Mohamed - 215830292

Yassin Mohamed - 213867064

December 3rd, 2020

Logs

To access real-time updated logs for application visit:
https://cloudwatch.amazonaws.com/dashboard.html?dashboard=SystemTracking&context=eyJSIjoidXMtZWFzdC0xIiwiRCI6ImN3LWRiLTI0MzMxNzI3ODYwNCIsIlUiOiJ1cy1lYXN0LTFfOFhHeWFVWldsIiwiQyI6IjJtMDQ0M2JqMWo0cGh1dGIzN2xqdTh1dG82IiwiSSI6InVzLWVhc3QtMToxOGM1NDAxOS1iZGFlLTRhZjAtYjZjYy1mYTgwNzdlNmNkY2YiLCJNIjoiUHViGljIn0%3D

Prototype link

http://systemtracking-env.eba-4npp3w82.us-east-1.elasticbeanstalk.com/login
Username: esra

Password: esra1

Abstract

This report starts by examining the evolving requirements of software engineering systems in modern society. It then details the process of implementing a self-healing loop following the MAPE-K architecture into the software system; more specifically, a logging and analyzing component that aims to determine the status of the system. It then details the changes to the existing software structure due to the implementation of the loop, and the effects it has on the software quality attributes. Finally, it details the interactions between the newly implemented system and the pre-existing one, using diagrams to illustrate the difference.

Introduction

As software systems become bigger, and more people need to access them, it becomes necessary to rethink the ways in which software systems work. In the current state of the world, entire chunks of the internet are controlled by a small number of Top-Level-Domains, meaning that these systems are responsible for managing hundreds of thousands of user requests to thousands of different servers and either serve them the desired service, or route them appropriately. In addition to their sheer size and complexity, these systems have also become critical in the functioning of societies around them. Things like online banking and inter-bank networks, government record databases and university online systems are crucial to the environments that house them, while being too large for people to constantly fox any problems that come up on their own, and too important to be offline for the duration that it would take. In response to these concerns, the principles of Autonomic Computing detail the approach to design needed to mitigate such issues.

Autonomic Computing is an approach to software design that ensures a software system will be able to operate with minimal human input or interference, it aims to solve the aforementioned problem of large and complex systems that are difficult to monitor and fox by hand, but that need to have minimal downtime. The approach details several abilities that an Autonomous, or Self-Adaptive, software system needs to have, that ensure it is able to maintain itself without input. The system needs to be able to: 1- Configure itself during runtime based on the context surrounding it. 2- Recover from a failed state without interrupting the application. 3- Optimize use of resources and allocation to meet utilization needs with minimal costs. 4- Ability to detect and protect itself against intrusive behavior. These principles are applied throughout complex software systems everywhere to ensure they can adapt to increasing loads, and fix errors on the fly without needing intervention from human programmers.

Concerning our system, the journey taken by our team to reach this point in the development cycle has been long. We started by conceiving of an idea and writing the proposal, through a conceptual design phase, and finally to a prototype implementation phase. During our team's time working on the concrete architectural report, we were concerned with bringing about our theoretical plans and architectural patterns to life through the implementation of the system. The process of researching the available platforms for development, as well as the different services and components available to us to use forced us to redefine our conceptual architecture many times over until we arrived at our final form. During our iteration phases, there were multiple instances where we were unable to achieve certain goals or attributes for our system, either due to constraints of the platform we were developing on or technical difficulties in applying our plans. Finally, we implemented our system based on our concrete architecture and adjustments to create the 0.1 prototype. In this phase, we are asked to add a self-healing loop in our system, by

implementing a Monitoring and Analyzing component to our system that is able to detect and notify users of errors based on the logs.

Enhancement

Overview

The use of a self-healing loop in our software system allows us to ensure our system achieves maximum uptime while reducing the occurrences of failure and service interruptions. This is achieved by giving our system the ability to monitor its own behavior, determine what errors have occurred, and take the necessary steps to ensure the system remains online. In order to give our system the ability to self-monitor and repair, we must implement the MAPE-K feedback loop. The MAPE-K model details the feedback loop that our system will go through to self-repair. The different phases of the feedback loop are:

Monitor: a component in the system monitors and collects real time data about the execution of your code.

Analyze: a component in the system uses the information provided by the monitoring component to determine the current status of the application, turning the raw monitoring data into actionable contextual information.

Plan: based on the analysis performed, the planning component determines what the best mitigation tactic is if any.

Execute: this component executes the mitigation tactics decided on by the planning component, it implements the changes in structure in run-time that allows the system to keep running without human interference.

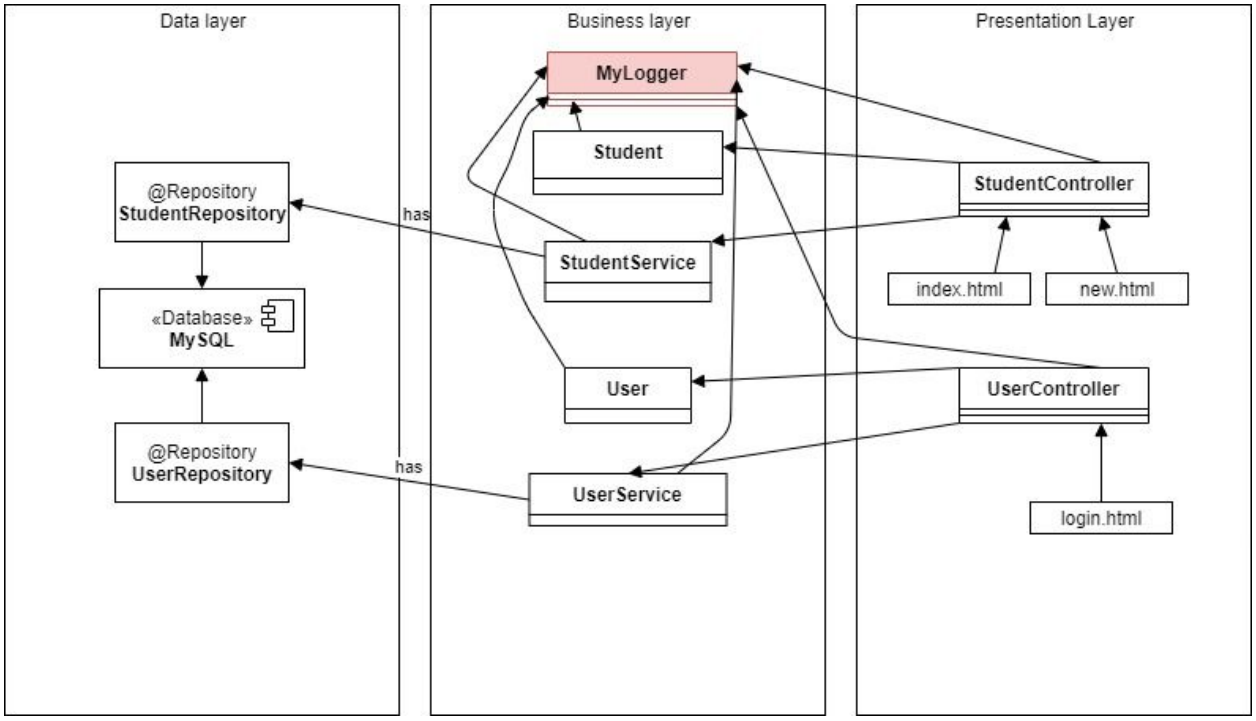
In our system, we're going to enhance our self-healing capabilities by implementing a logging and analysis component. Since our system has many interconnected parts communicating both remotely through HTTPs protocol and internally through API calls, there are many possibilities for failures. In addition, in the case of a system failure, the development team wouldn't have any information by default as to where the error occurred, and would have to approach the solution reactively, which means the system would be down until further notice. Adding a logging component to our system allows us to trace the execution of the code up to the point of failure, with the logs giving us enough information to be able to determine which service failed, or which method call threw an exception, and at what time. This gives the development team much needed information about the specific services or components that were involved in the failure.

Furthermore, adding an analysis component allows us to use the information gathered by the logging component to examine the current status and context of the system, and determine if there are any faults in the execution. The addition of the analysis component increases the effectiveness of our logger, as it turns the raw log data into the relevant contextual information necessary for a MAPE-K model.

Impact on Architecture

On a high-level, our system implements a Layered software architecture composed of three layers, the Presentation, Business, and Data layers. The defining feature of the layered architecture is the existence of separate layers that each only interacts with the ones adjacent to it. The addition of the logging and analysing components to our system necessitated some changes to the structure and architecture of our program, however, since the added components still abided by the constraints of our existing Layered architecture, the resulting enhanced system still followed the same software architecture. To implement the logging and analysing function we had to add a new component to the Business layer of our application. This component communicates with other components in the Business layer, in order to log and analyze any errors in the intermediate steps of our system, and with components in the Presentation layer, where it logs user actions and ensures fault-free execution of the interface. The resultant system still abides by the constraints of our original architecture, and so affords us the same advantages that it does. In addition, we added an additional View to the presentation layer through which users can view the log and analyzer output, this is both for remote monitoring and DevOps cooperation.

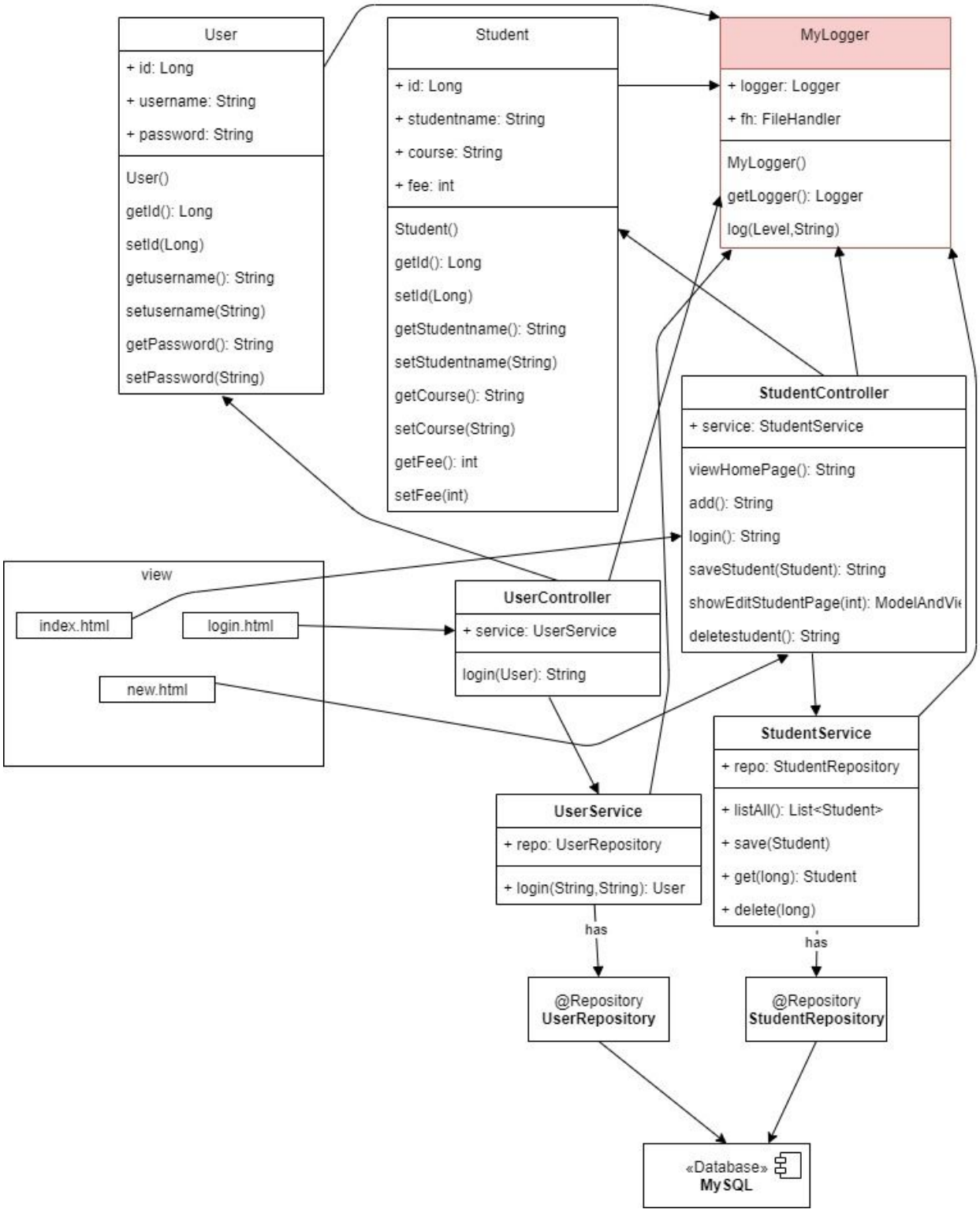
Architecture Diagram (w/ enhancement)



Interactions with existing components

For our Logging and Analysis component to be as effective as possible, it has to monitor the execution of the program on all the different layers, to that end different actions and method calls throughout the layers of our application trigger our logging component. In the presentation layer, the logging component is triggered at key points in the execution to ensure adequate information about the state of the front-end is conveyed to the analyser. Additionally, in the business layer, the logger gets triggered on the start and end of key operations in our logic, ensuring the analyser has information regarding the creation of internal objects and any exceptions that would be thrown.

Class Diagram (w/ enhancement)



Concurrency

Our system is deployed on AWS, which features built in features that ensures any number of users are able to access the system at any given time. When configuring the deployment of the application, we specify how many instances exist at the start and how many can be created later. Additionally, we enabled a Throttling system that ensures clients won't bombard the server with requests and crash it. Finally, we enabled AWS's load balancing feature, which dynamically creates instances and allocates resources to the system for optimization purposes.

Effects on Quality Attributes

The addition of the Logging and Analyzing components to our system has a notable impact on how our software meets the target quality attributes. In some cases, it enhances a given attribute or helps the system meet it, while in others it introduces new risks and might even disqualify them.

Maintainability: The Logging and Analyzing component provides developers with information on how the system is running and what types of errors happen most frequently. This means the development team is always aware of the state of the system during any problems and can work much faster to address them, additionally, it provides the development team with information and statistics it can use to understand the type of load the system is undergoing and take preventative measures to deal with them.

Evolvability: Similarly to maintainability, the enhancement in this case provides developers with knowledge of the trends and patterns of use of their system, this allows them to more accurately decide on what new features to add or changes to implement. However, it is important to note that any new components added to the system will have to be connected to the logging component, which adds a small amount of overhead.

Testability: The enhancement provides crucial information about the state of the software during runtime, these features can be leveraged when testing the software system to perform more accurate and effective tests.

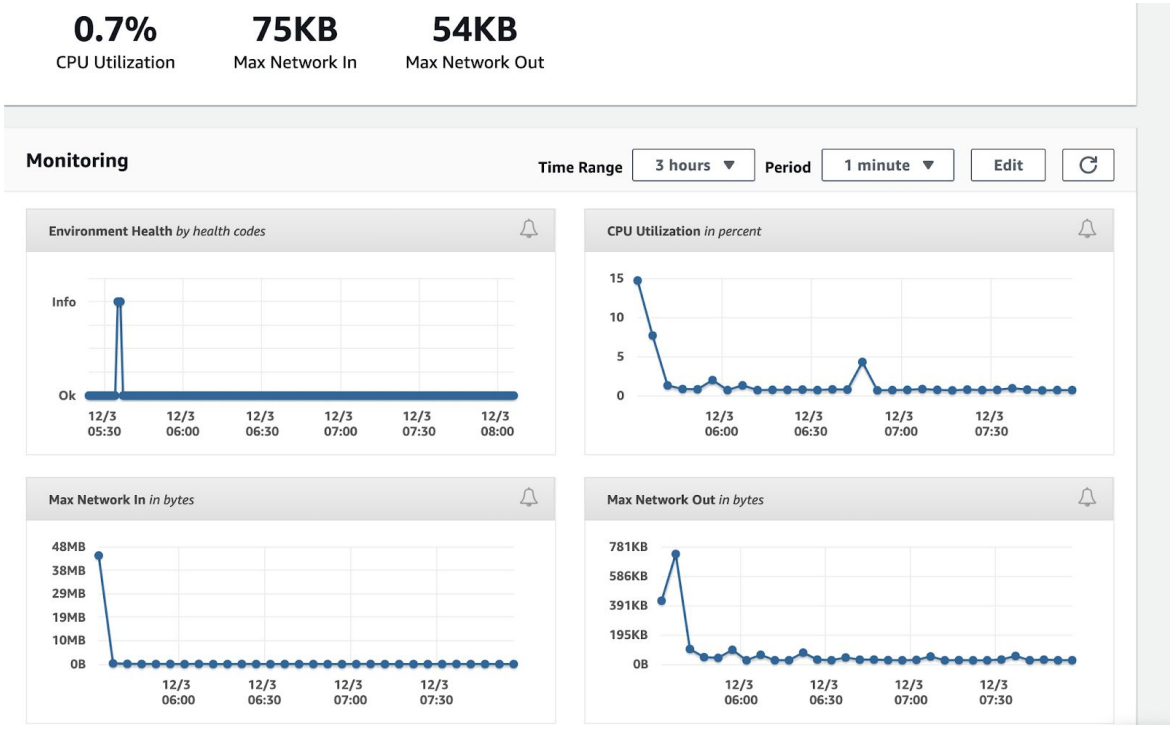
Performance: The information provided by the logger, especially relating to server response time, provides crucial metrics for determining the best resource optimization path for the system. Note that the addition of the logger and analyzer adds extra computation steps, but they have not been found to impact the performance of the system negatively.

Security and Privacy: The creation of an extensive logger brings with it the risk of leaking critical implementation details or private information, this increases the probability of malicious attacks if information gets used in attacks against our system.

Alternatives for implementation

When deciding on the specific implementation of our logging function, we were faced with multiple different logging classes that work with our Spring application. The first option in front of us was the default spring logger that is built-in to the framework. Initially, we looked at the information provided by the default spring logger and compared it to the monitoring capabilities we wanted our system to have; we found that while the default logger outputted a lot of information relating to the execution of the underlying framework, it naturally gave no details about the implementation of the business or presentation layers or the state of our classes and objects during runtime. Additionally, it provided limited flexibility with regards to customizing the log output and location. We then examined the logging features provided by Amazon Web Services (AWS), on which we deployed our application. We found that AWS provides extensive logging features that relate to a large portion of our system, such as logging the running instances of the container and established connections to clients. AWS also provides visual metrics that would be crucial to improving the resource utilization of our system.

Visual metrics provided by AWS



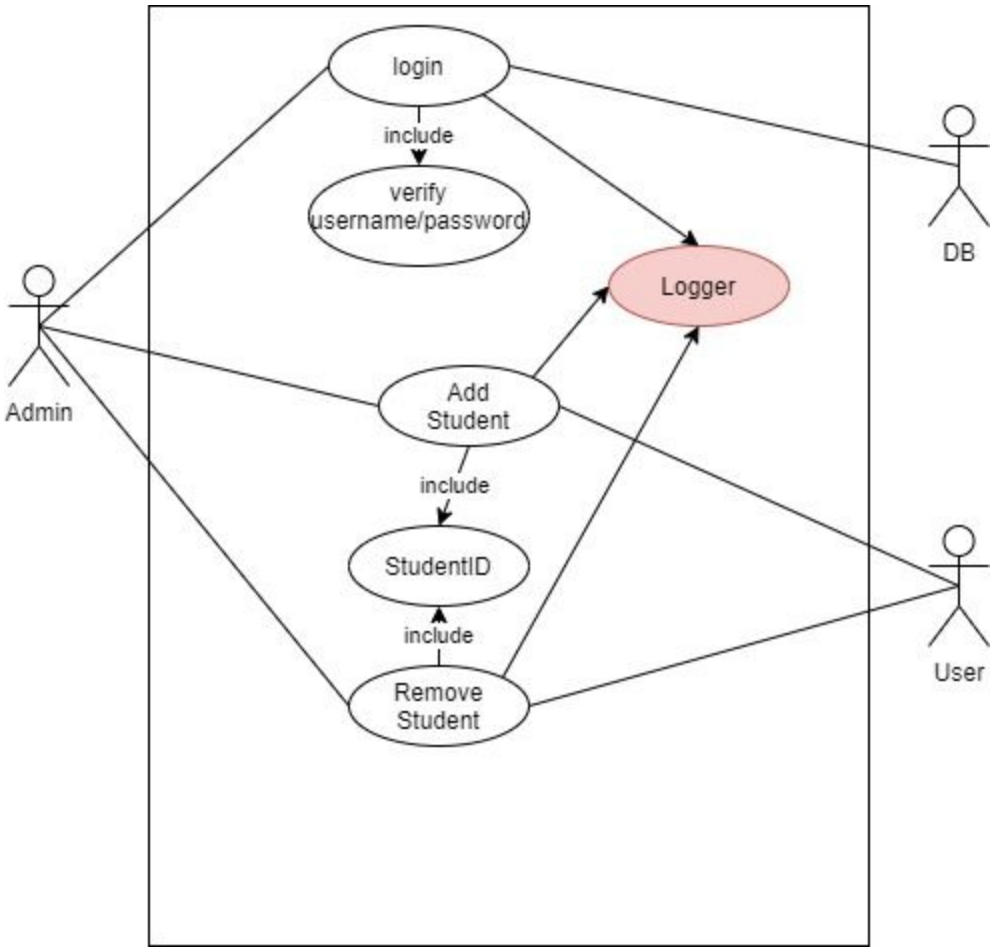
Unfortunately, we realized that the AWS logging functions, while much closer to something we can use for analysis, still only pertained to specific components of the system, and did not allow us to extend it to our implementation. In the end, we decided to extend the capabilities of the default spring logger by combining it with the Java Logger API, giving us access to its features and methods, while allowing us to still access the logs generated by the framework. This allowed us to make use of the existing logging features and enhance it with our own, and let us use both sources of information for the Analysis step.

Plans for testing

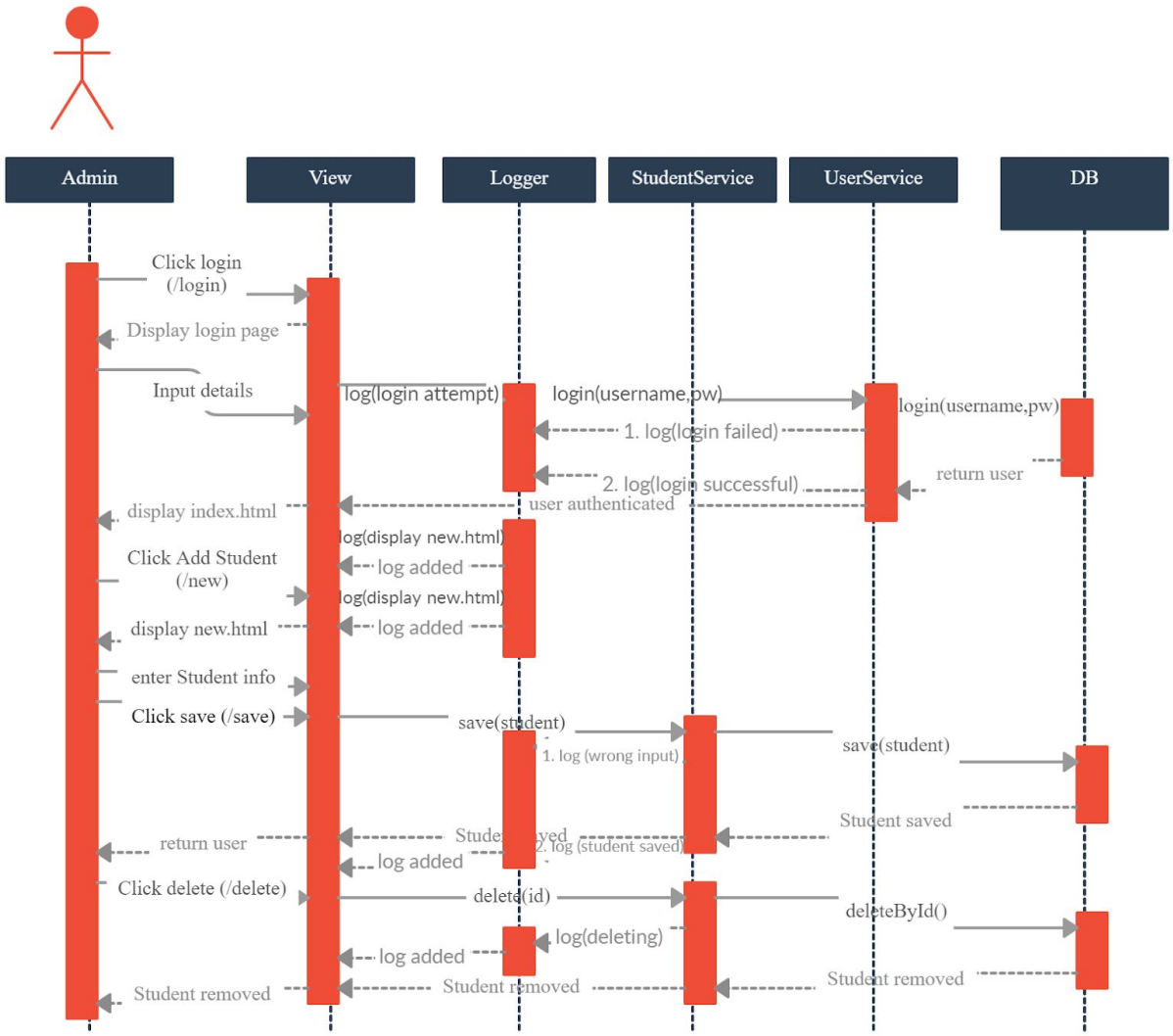
To ensure our logging and analysis components are working correctly, we will use fault injection to test how our system responds to errors and unforeseen problems. Particularly, we will be looking to trigger faults in the system and checking that the logger collects the information correctly, and that the analyzer component is able to identify the problem. These faults would be things that might occur naturally in the lifecycle of the program, such as service interruptions and incorrect parameters.

Software Diagrams (w/ enhancement)

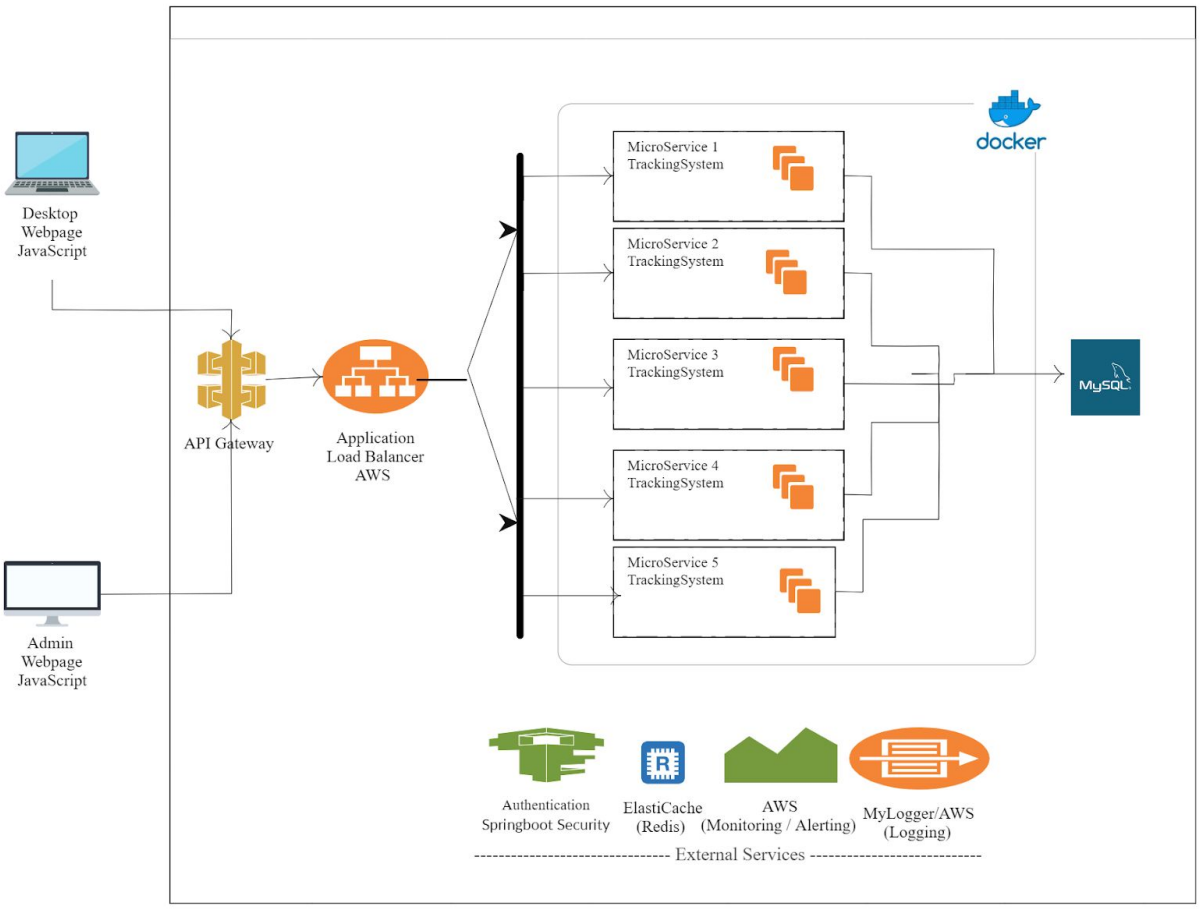
Use case diagram



Sequence diagram



Component diagram (w/ load balancer)



Conclusion

During this development phase we realized through practice the importance of autonomous computing systems, as with the increase in the number of people using web applications, as well as the increasing size and complexity of software systems, it becomes infeasible to manage and repair the system by hand. We've seen the impact that a logging function can have in the continuing health of a software system, by offering the developers more information on the state of the software and context around when faults are being triggered. These factors exemplify how certain approaches to software development can have a crucial role in the feasibility of a certain system.

Lessons Learned

During this design phase, we realized the importance of building your system while accounting for self-healing loops from the start, this would've helped us implement more extensive Planning and Mitigation tactics.