****

# CS 305 Project Two

**Practices for Secure Software Report**

Table of Contents

[Document Revision History 3](#_Toc33111302)

[Client 3](#_Toc33111303)

[Instructions 3](#_Toc33111304)

[Developer 4](#_Toc33111305)

[1. Algorithm Cipher 4](#_Toc33111306)

[2. Certificate Generation 4](#_Toc33111307)

[3. Deploy Cipher 4](#_Toc33111308)

[4. Secure Communications 4](#_Toc33111309)

[5. Secondary Testing 4](#_Toc33111310)

[6. Functional Testing 5](#_Toc33111311)

[7. Summary 5](#_Toc33111312)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **4-17-2021** | **Daniel Tipton** |  |

## Client



## Instructions

Deliver this completed Practices for Secure Software Report documenting your process for writing secure communications and refactoring code that complies with software security testing protocols.

Respond to the steps outlined below and replace the bracketed text with your findings in your own words. If you choose to include images or supporting materials, be sure to insert them throughout.

## Developer

Daniel Tipton

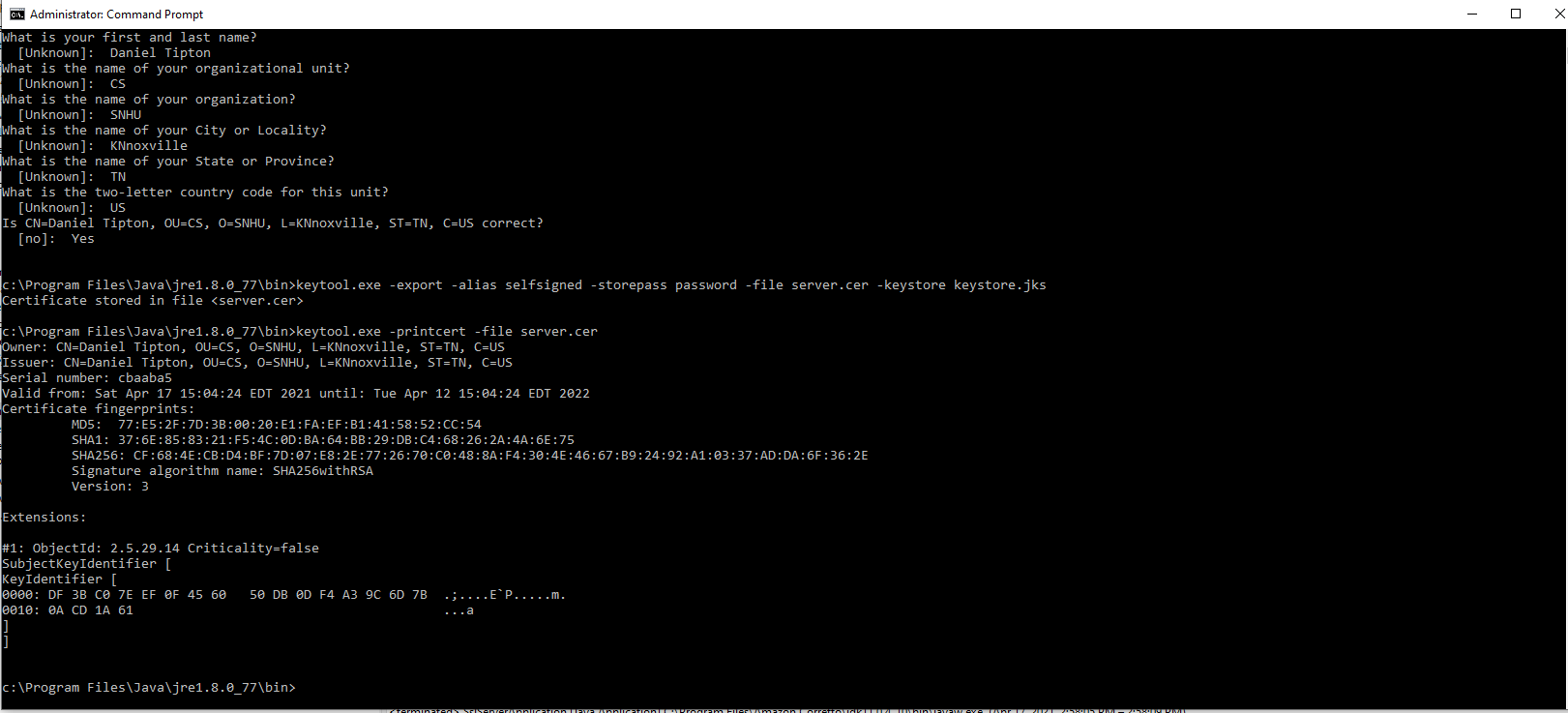
## 1. Algorithm Cipher

The choice of an algorithm cipher is a tough one. Especially as cyber security becomes more and more key to a company’s success in the 21st century. Artemis Financial needs an algorithm that is going to be a secure as possible. With most of the files being archives the speed of the encryption isn’t as big of an issue with I recommend SHA-256 encryption. This is the most secure form of encryption available with base java with 256 big encryptions. This means that the key for the data is 256 bits long, which to a human doesn’t sound too big, but when its all randomized it pretty much prevents most of all attacks. Since the data won’t be moved much, we can use the symmetrical 256 encryption and only must worry about one key as opposed to the having a public and private key.

The idea of having encrypted data isn’t a new invention by any means there have been attempts to conceal data for as long as there has been written communication, more modern encryption can be trace back to the use of “enigma” machines during world war 2 that the Germans used to hide communications. Cracking the code was one of the keys to defeating the Nazis for the British.

## 2. Certificate Generation

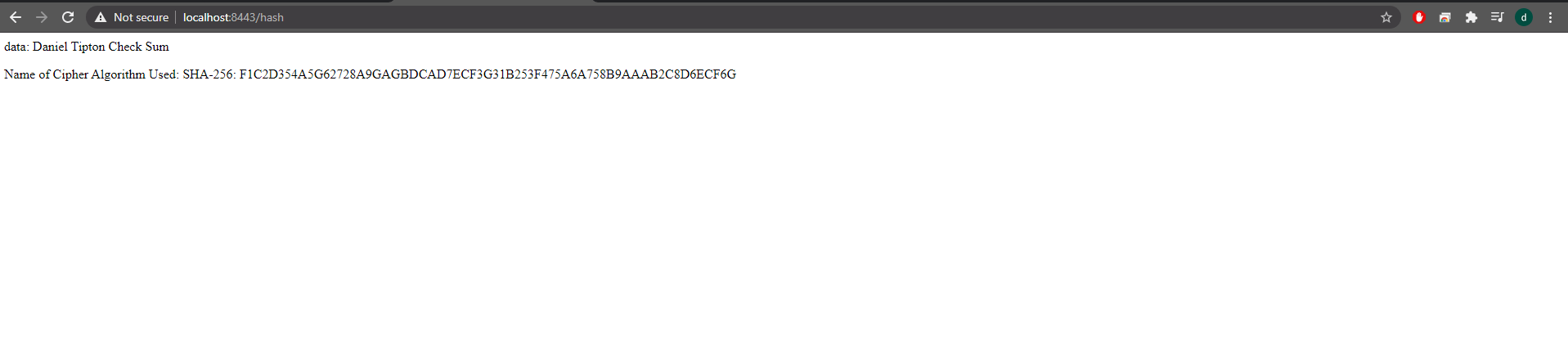
Generate appropriate self-signed certificates using the Java Key tool, which is used through the command line.



## 3. Deploy Cipher

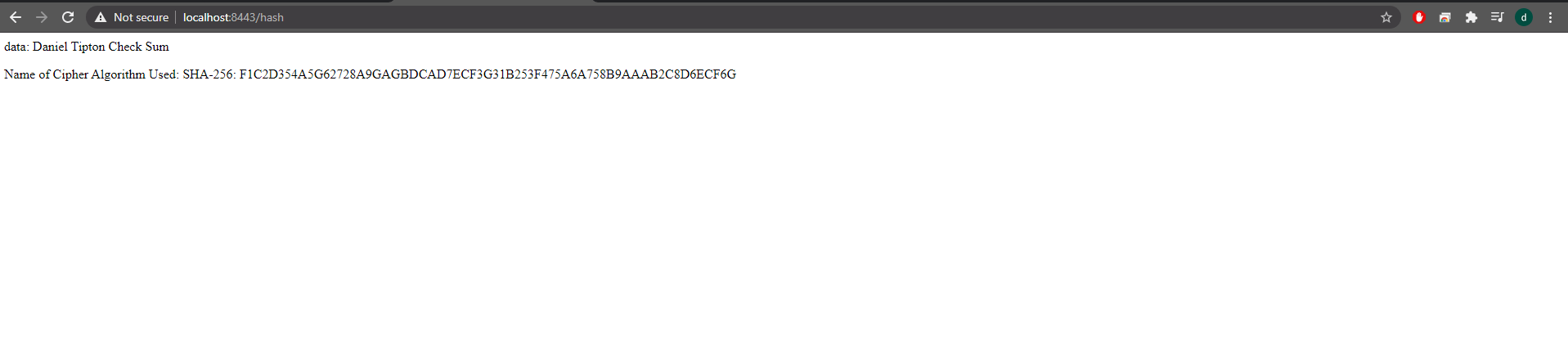
Refactor the code and use security libraries to deploy and implement the encryption algorithm cipher to the software application. Verify this additional functionality with a checksum.

* Insert a screenshot below of the checksum verification. The screenshot must show your name and a unique data string that has been created.



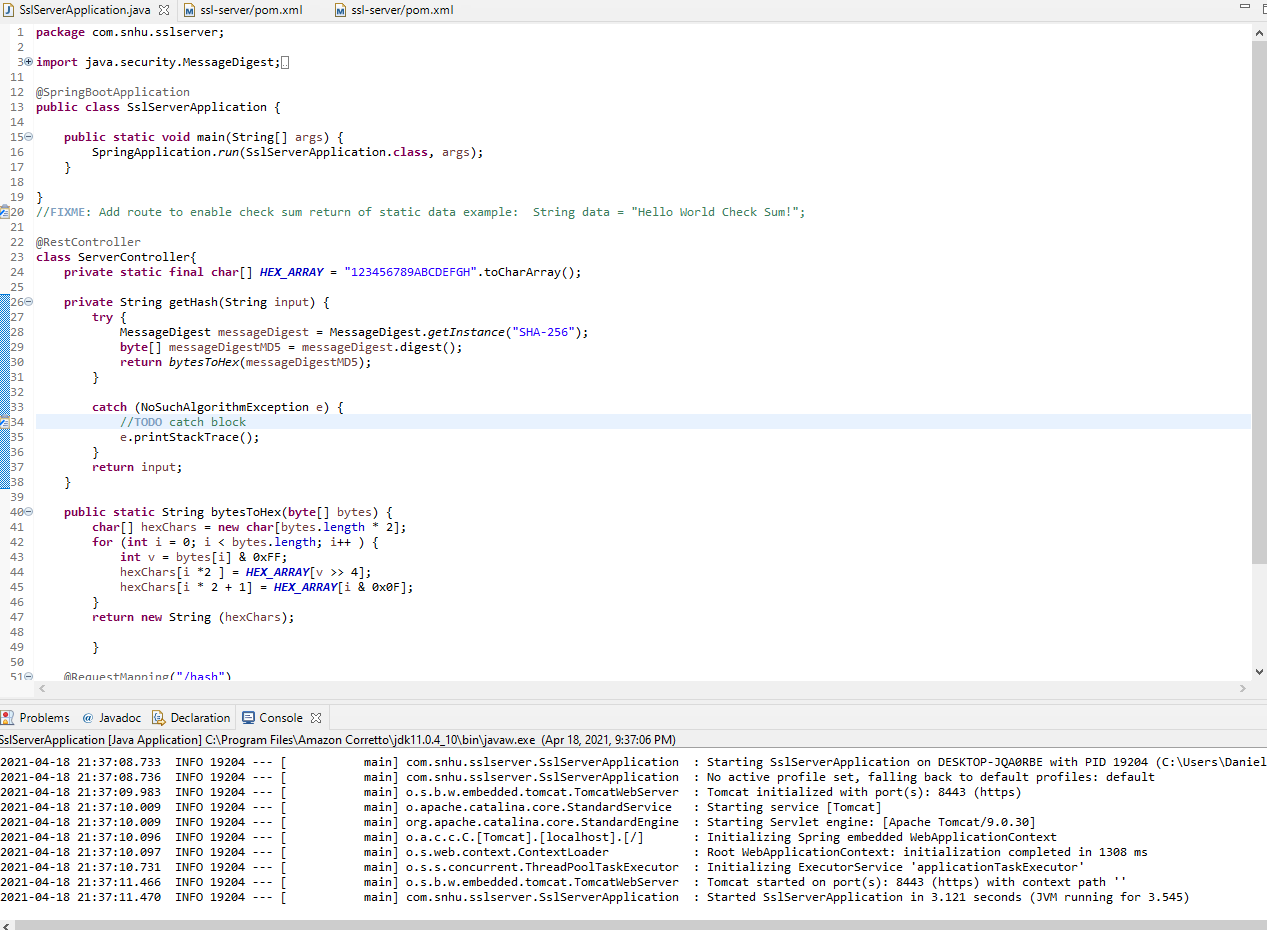
## 4. Secure Communications

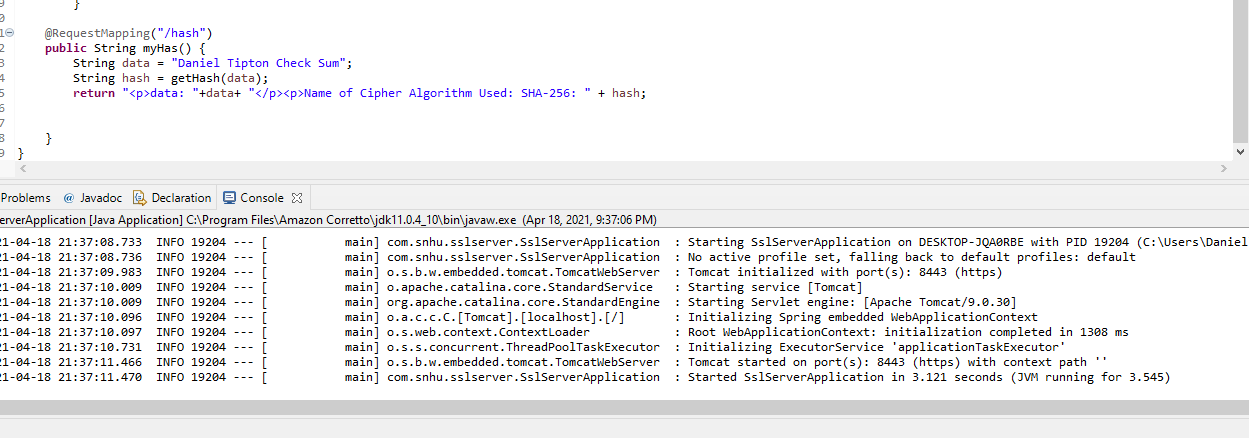
Refactor the code to convert HTTP to the HTTPS protocol. Compile and run the refactored code to verify secure communication by typing **https://localhost:8443/hash** in a new browser window to demonstrate that the secure communication works successfully.

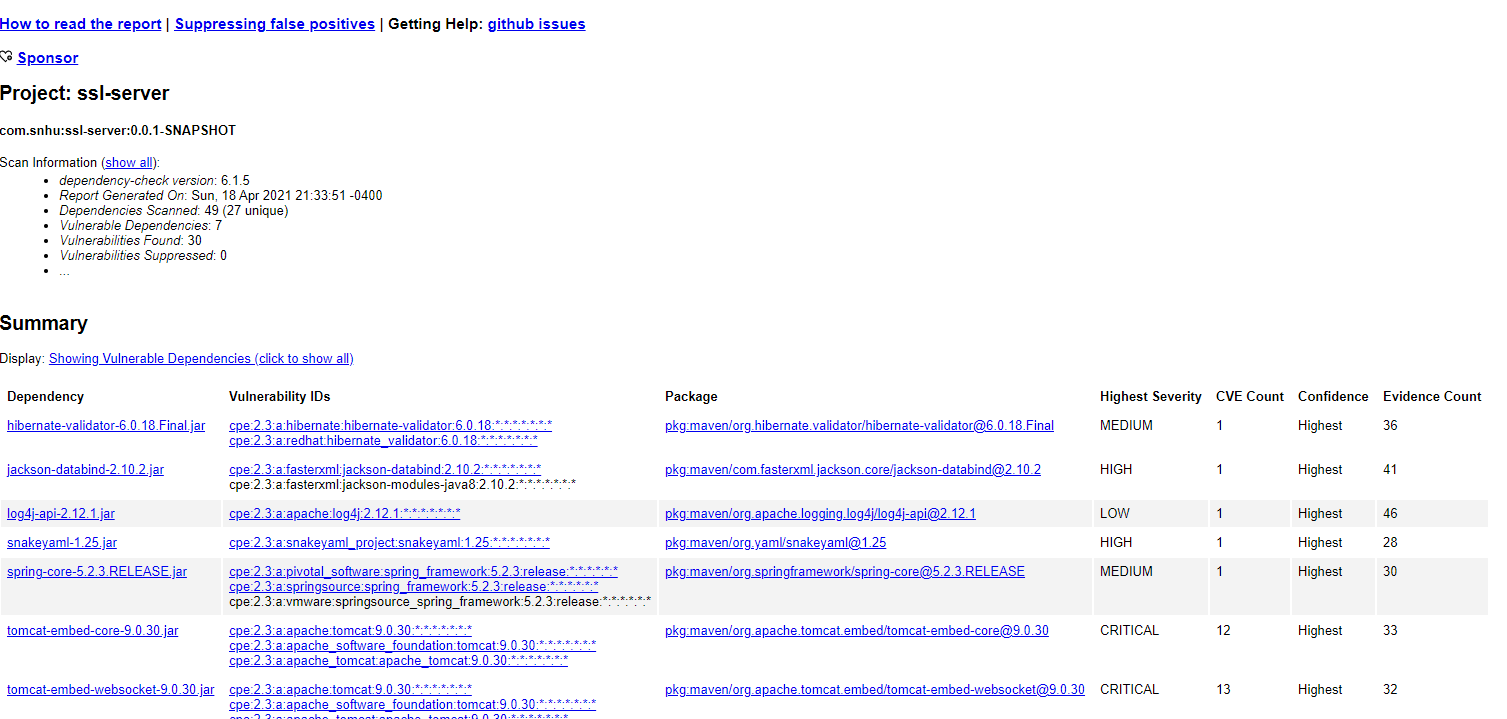


## 5. Secondary Testing

Complete a secondary static testing of the refactored code using the dependency check tool to ensure code complies with software security enhancements. You only need to focus on the code you have added as part of the refactoring. Complete the dependency check and review the output to ensure you did not introduce additional security vulnerabilities.



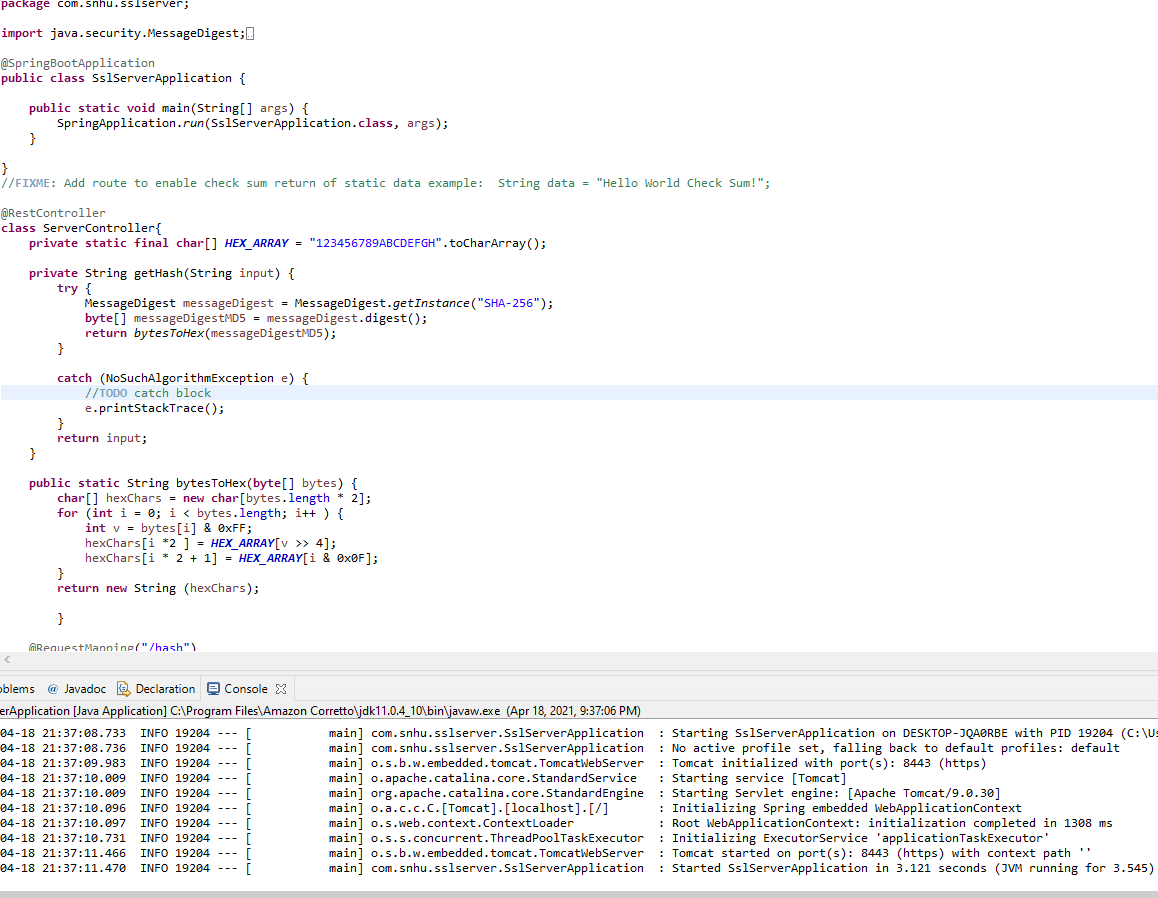


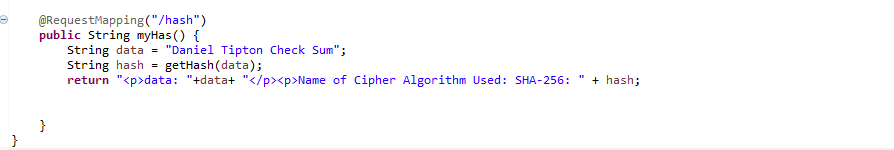


## 6. Functional Testing

Identify syntactical, logical, and security vulnerabilities for the software application by manually reviewing code.

* Complete this functional testing and include a screenshot below of the refactored code executed without errors.





## 7. Summary

Discuss how the code has been refactored and how it complies with security testing protocols. Be sure to address the following:

I’ve added additional security in the form of a hashing function. This will allow the application to adhere to secure coding via the rest controller class called server controller. This encrypts it with 256-bit encryption allowing for the most secure data outside of social engineering allowing a user into the system. My process to adding layers of security was to first look at the aim of the software and the type of information it contains and the functions that it should provide. From there I check for any points of entry that could allow someone to access private information or get unauthorized control of a system. By going through a well-defined process, a company can make safe and secure coding a primary goal, and not a knee jerk reaction to a compromised system. For this software checking the dependencies often and making sure that the checker is up to date (I had to update to 6.1.5, so it was a tad bit out of date). Keeping the key for this data as secure as can be (Other than a crazy lucky brute force the only real way to get in is through some social engineering). Those would be my two big things to watch for.