****

# Practices for Secure Software Report

Table of Contents

[Document Revision History 3](#_Toc102040754)

[Client 3](#_Toc102040755)

[Instructions 3](#_Toc102040756)

[Developer 4](#_Toc102040757)

[1. Algorithm Cipher 4](#_Toc102040758)

[2. Certificate Generation 4](#_Toc102040759)

[3. Deploy Cipher 4](#_Toc102040760)

[4. Secure Communications 4](#_Toc102040761)

[5. Secondary Testing 4](#_Toc102040762)

[6. Functional Testing 4](#_Toc102040763)

[7. Summary 4](#_Toc102040764)

[8. Industry Standard Best Practices 4](#_Toc102040765)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **June 28th, 2025** | **Eric Schooler** | **Edit** |

## Client



## Instructions

Submit these completed practices for a secure software report. Replace the bracketed text with the relevant information. You must document your process for writing secure communications and refactoring code that complies with software security testing protocols.

* Respond to the steps outlined below and include your findings.
* Respond using your own words. You may also choose to include images or supporting materials. If you include them, make certain to insert them in all the relevant locations in the document.
* Refer to the Project Two Guidelines and Rubric for more detailed instructions about each section of the template.

## Developer

Eric Schooler-Davison

## Algorithm Cipher

For this project, data vulnerability is a big problem as we are dealing with communications between A and B where key handling is very important. Of course, we could use AES as it is generally a gold standard for encryption; however, AES is symmetric, and we want to use an asymmetric key handling system to avoid interception of the decryption keys. For this, we will use RSA. Unlike AES, RSA generates both a public key and a private key. This makes the process slower but has the added benefit of being more secure when it comes to key exchanges. When you encrypt data, you do so with their public key, then they decrypt it with their private key. This way data can only be opened by the intended recipient, and the sender does not know the recipient's private key and doesn't need to. This solves our key problem.

Next, we will be determining the hash function that we will be using. In this case, we will use SHA-256 for more secure handling unlike our certificate, which includes SHA-384 with RSA encryption. The point of SHA is to create a fingerprint of the data in the form of a hash. This isn’t used for the actual data you are receiving and is instead called upon to give a unique ID to the sent data so that the recipient can verify the data integrity and ensure that no one has injected anything into or tampered with it. When you receive the data, you generate another hash and compare the two. If they are the same, then the data has not been changed.

Encryption algorithms are used based on the use case. Historically, encryption has existed for a long time in the form of cryptography for words where you would create a cypher such as moving each letter in the alphabet over by one letter, then doing so in reverse to get the original back. From then DES was created and used as a gold standard, until modern computing proved to be strong enough to require new innovation. It was then replaced by AES and asymmetric key algorithms like RSA. Now encryption is used to secure sent data as well as stored and has many forms. The future will likely require more innovation soon, as quantum computing has the theoretical power to brute force these algorithms. This change may bring about obsolescence as encryption will require more powerful keys to protect against intrusion, but also more powerful devices to interoperate these encryptions.

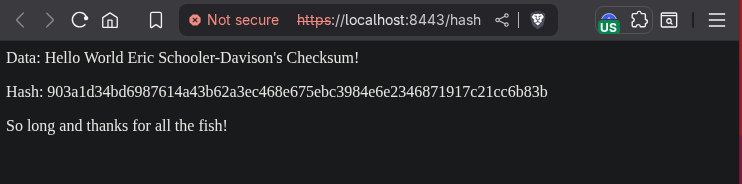
## Certificate Generation

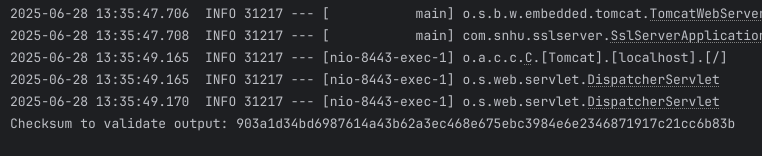
Insert a screenshot below of the CER file.



## Deploy Cipher

Insert a screenshot below of the checksum verification.

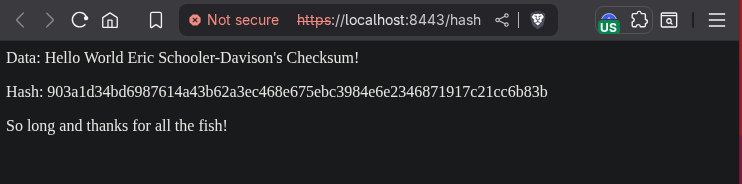




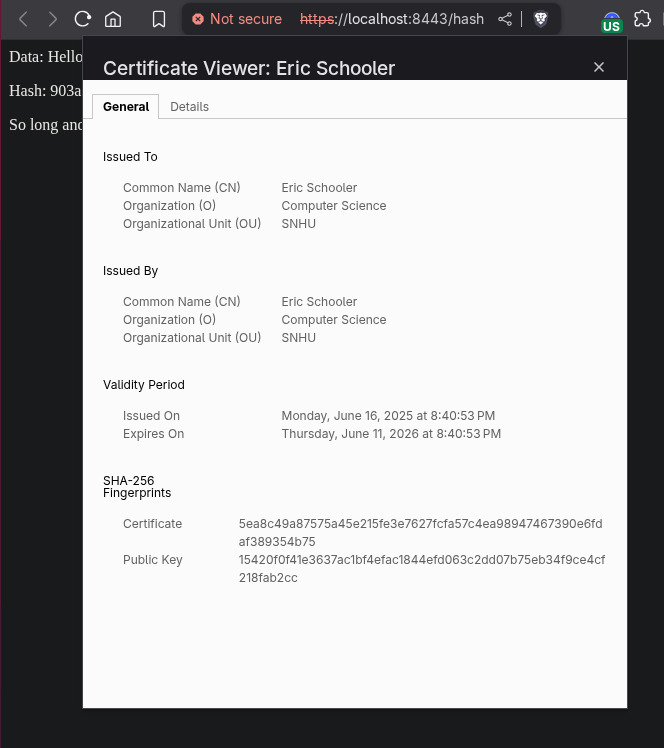
Picture 1 is of the web browser's output on the recipient's side. Picture 2 is of the terminal in the IDE listing the original checksum on the sender's side.

## Secure Communications

Insert a screenshot below of the web browser that shows a secure webpage.

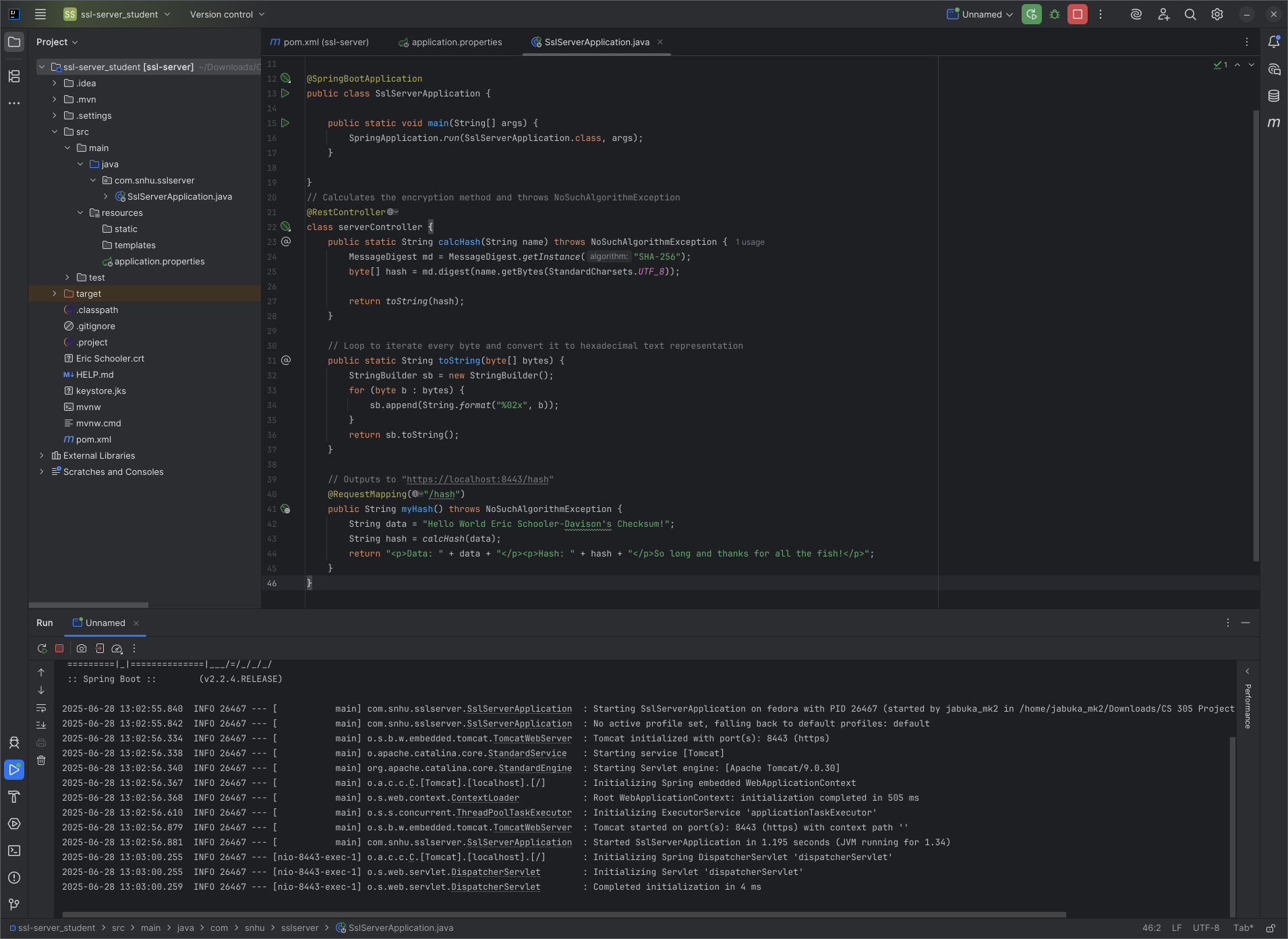


Something to note is that the browser lists the connection as not secure, I have validated that the certificate is validating properly, and the connection is encrypted. The browser uses a list of trusted CA’s to ensure self-signed certificates aren’t used. I started to fix this and convince my OS by uploading the server.cer file to my trust store in my root directory. However, I figured that this was beyond the scope of this course and left it as is. Just in case I have included a picture of the certificate that generates with the web address.



## Secondary Testing

Insert screenshots below of the refactored code executed without errors and the dependency-check report.



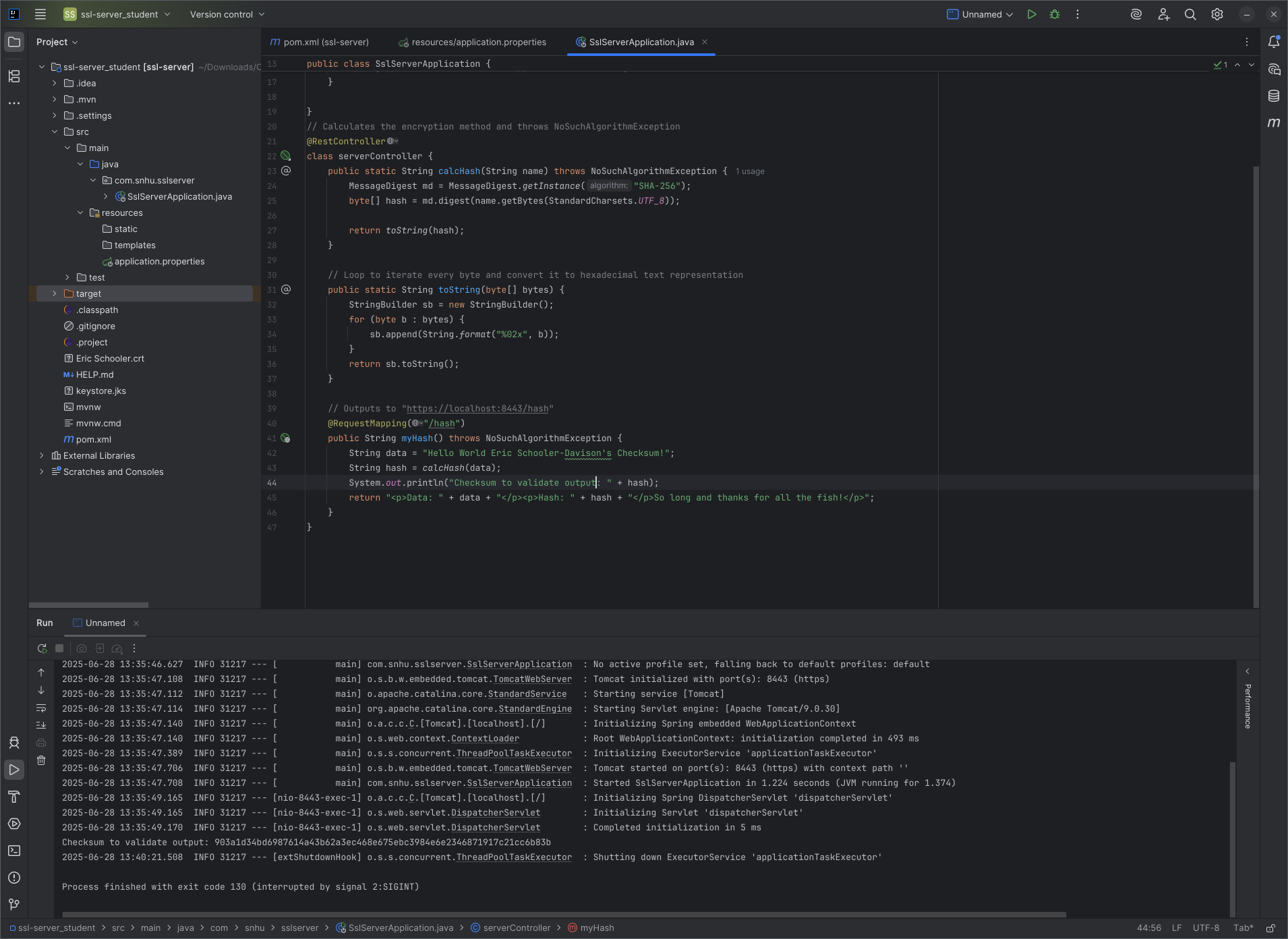
Here is a picture of the dependency check running before my code edits and after. As you can see, the number of vulnerabilities went up by 3. I have validated that all 3 new vulnerabilities are due to CVE updates completed from when I first tested it on the 16th to today the 28th.





## Functional Testing

Insert a screenshot below of the refactored code executed without errors.



## Summary

In the edits made in this document, we addressed several vulnerabilities in this code. The first workflow we addressed was secure API interactions. To have secure API interactions, you need things such as HTTPS TLS encryption and certificate authentication. We used cryptography in the certificate delivery which was in RSA, and the hash cryptography method SHA-256 for validation. We need this validation to ensure files are not tampered with in transit. We secured client/server interaction by using RSA, which uses public keys and private keys so that the client/server doesn’t have to expose keys when transferring them to the receiver. We had secure error handling handled for us by integrating with the REST API and Spring systems. Lastly, we refactored the code, designing it to be secure to prevent attacks such as injections. We used Maven dependency check to identify vulnerabilities that were created by our newly introduced code. In my project I compared my before and after to see if any variability existed after the new code was introduced. Many vulnerabilities were identified from the outdated version of applications such as spring and maven, and our out-of-date JDK 1.8. We did not practice encapsulation any more than usual by using proper conventions and public/private classes.

For my code edits I included 3 parts. The first was the overall class serverController which had 3 functions for calcHash, toString, and myHash. calcHash took our string as input and threw the NoSuchAlgorithmException. It then generated a message digest with instance SHA-256, which was our hash algorithm. Then we parsed our bytes to a new variable hash digesting the data using charset UTF\_8. We then returned the hash variable to the toString function. The toString function would take the bytes input and format them to a hexadecimal text representation for each byte, then return the string using StringBuilder function. Last, we use RequestMapping to output the data to /hash. This function also declares the data and hash as outputs on the site.

## Industry Standard Best Practices

I used best practices by using proper naming conventions, whitespace, and comments to make the code easy to read and understand. We wanted to make use of the already existing parts of the program and only alter as much as we needed to. I would update the versions of everything and run further dependency checks, but I believe that may be out of scope for this project. I originally updated maven to the newest version, however, JDK 1.8 had compatibility issues, so I left it at 5.3.0. We also created a self-signed certificate to upgrade our connection to HTTPS from HTTP, enhancing our security. By creating secure code, we ensure that security is maintained from the top down. By understanding the vulnerabilities inherent to code, and the methods of attack, we can create thoughtful code that is written to mitigate those issues. This saves both time and money as fixing something after the fact is far harder than doing so when you're writing it. Some things such as buffer overflow and SQL injection are preventable. Writing insecure code can cost quite a bit to a company, ranging from DOS attacks which stop income to more malicious attacks that lead to data breaches if not properly accounted for. To my mind, one major issue is that the company generally doesn’t understand how the app they use works and lean toward a “if it ain’t broke” mentality. This leads to companies being far out of date, and vulnerabilities to be well known and easy to exploit. Innovation is not a burden, sure growing pains are going to happen, but a secure system will pay dividends.