

# 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** | **2/24/2024** | **Joy Kalinowsky** | Attached with this report are the code files, contained in a zip folder titled “Kalinowsky CS 305 Project Two Code Base”. |

## Client



## Instructions

Submit this completed practices for 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

Joy Kalinowsky

## Algorithm Cipher

The encryption algorithm cipher, SHA-256, is a hashing algorithm that uses a 256-bit key to convert a string of data or plaintext into an encrypted data string that is 256 bits in size. This is done by taking the string of data and converting it into binary data, which then has many iterations of compression functions applied to it. This results in hash values generated from each round. The hash value from the final round forms one final 256-bit hash value.

The purpose of the cipher’s hash functions and bit levels is to ensure data integrity, secure against unauthorized modifications, protect stored passwords, and operate at different speeds to suit different purposes.

Random numbers are used in cryptography for various applications, such as key generation, nonces, and salts in certain signature schemes, such as ECDSA and RSASSA-PSS.

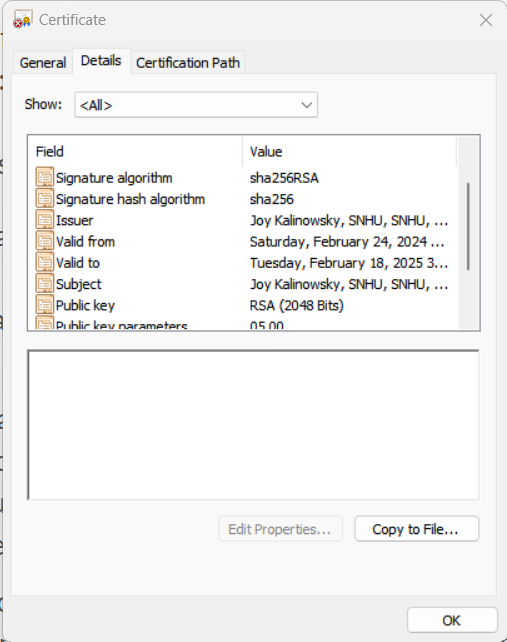
Symmetric keys are used in symmetric key algorithms. These algorithms use the same cryptographic keys for both the encryption of plaintext and the decryption of ciphertext. Symmetric key algorithms are used because it is inexpensive to produce a strong key for both ciphers to use.

Non-symmetric keys are used in asymmetric key algorithms. These algorithms use non-symmetric keys for digital signature authentication. Non-symmetric keys are used because they do not require securing key distribution channels or the exchange of private keys in contrast to symmetric keys.

Encryption algorithms have preceded the existence of computers. They were used as far back as 600 BC when the ancient Spartans invented a device called a scytale to send secret messages in battle. Encryption techniques have gradually improved throughout history. Some examples of historical ciphers include the Caesar cipher developed in 60 BC and the Bombe Machine designed by Alan Turing during World War II. In modern cryptography, ciphers are used to encrypt data at rest and in transit across the internet. Different types of encryption solutions are used to encrypt data online. Popular solutions include the Advanced Encryption Standard (AES), Public Key Infrastructure (PKI), and Elliptic-curve Cryptography (ECC).

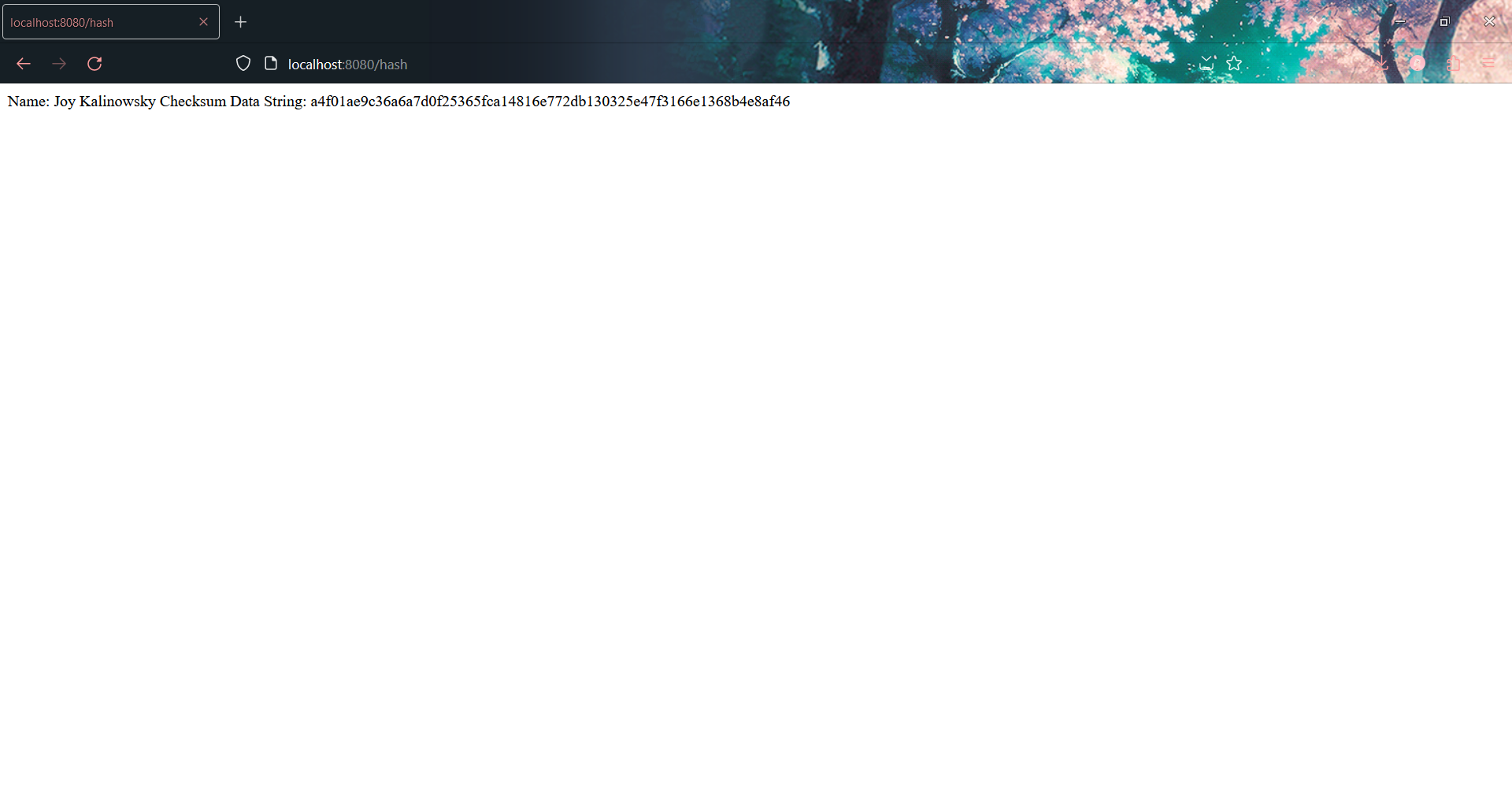
## Certificate Generation

Insert a screenshot below of the CER file.



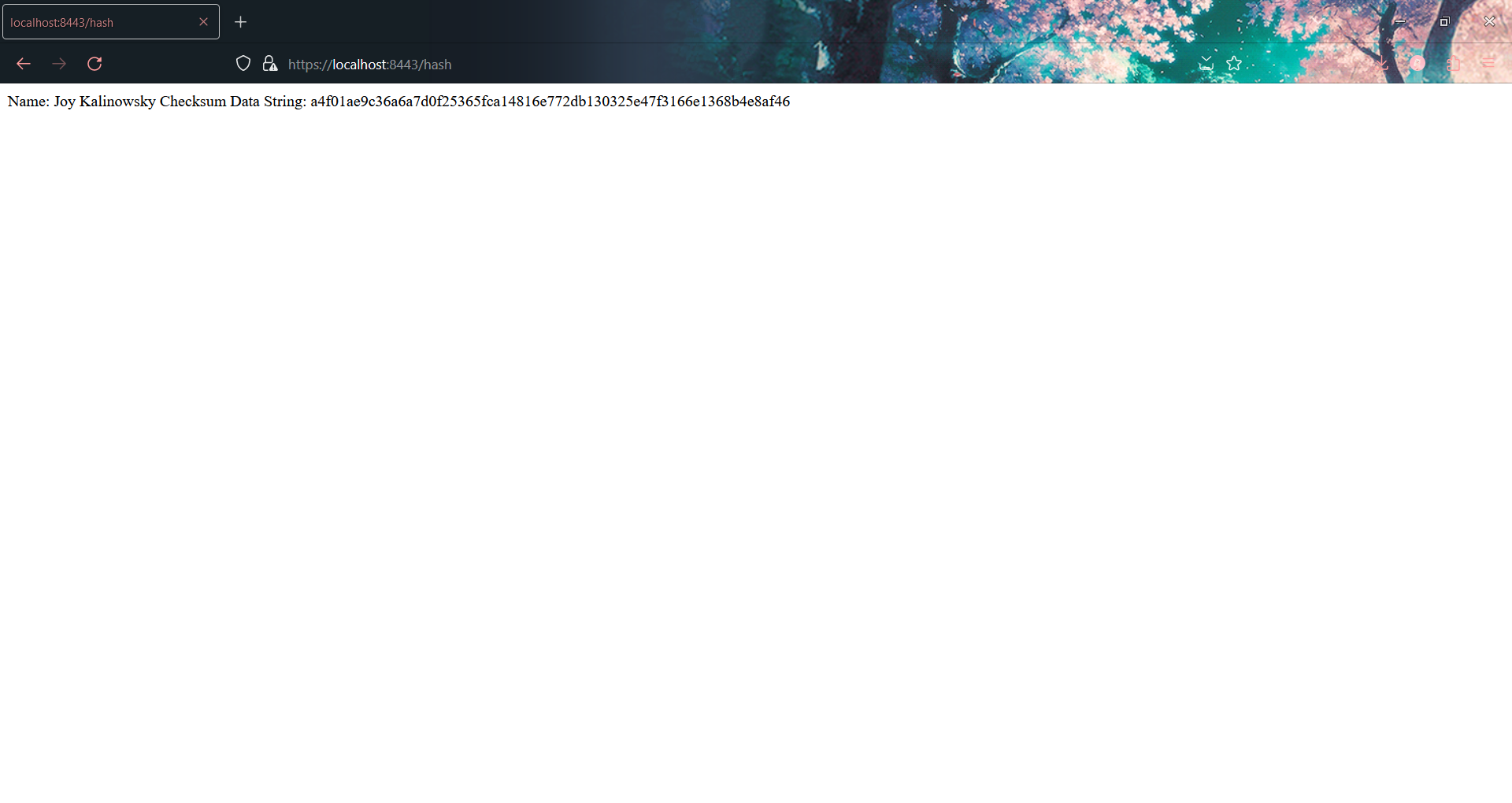
## Deploy Cipher

Insert a screenshot below of the checksum verification.



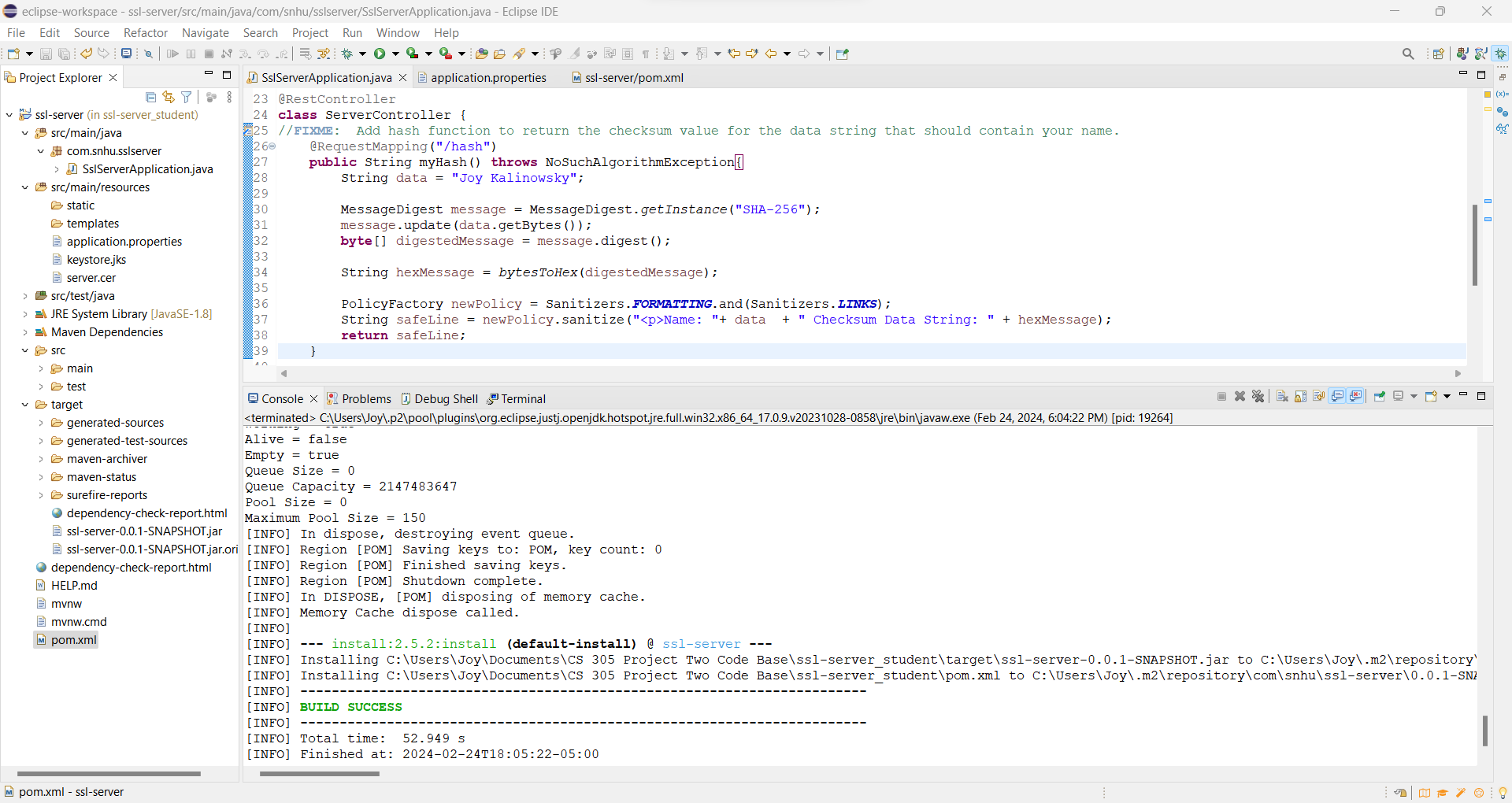
## Secure Communications

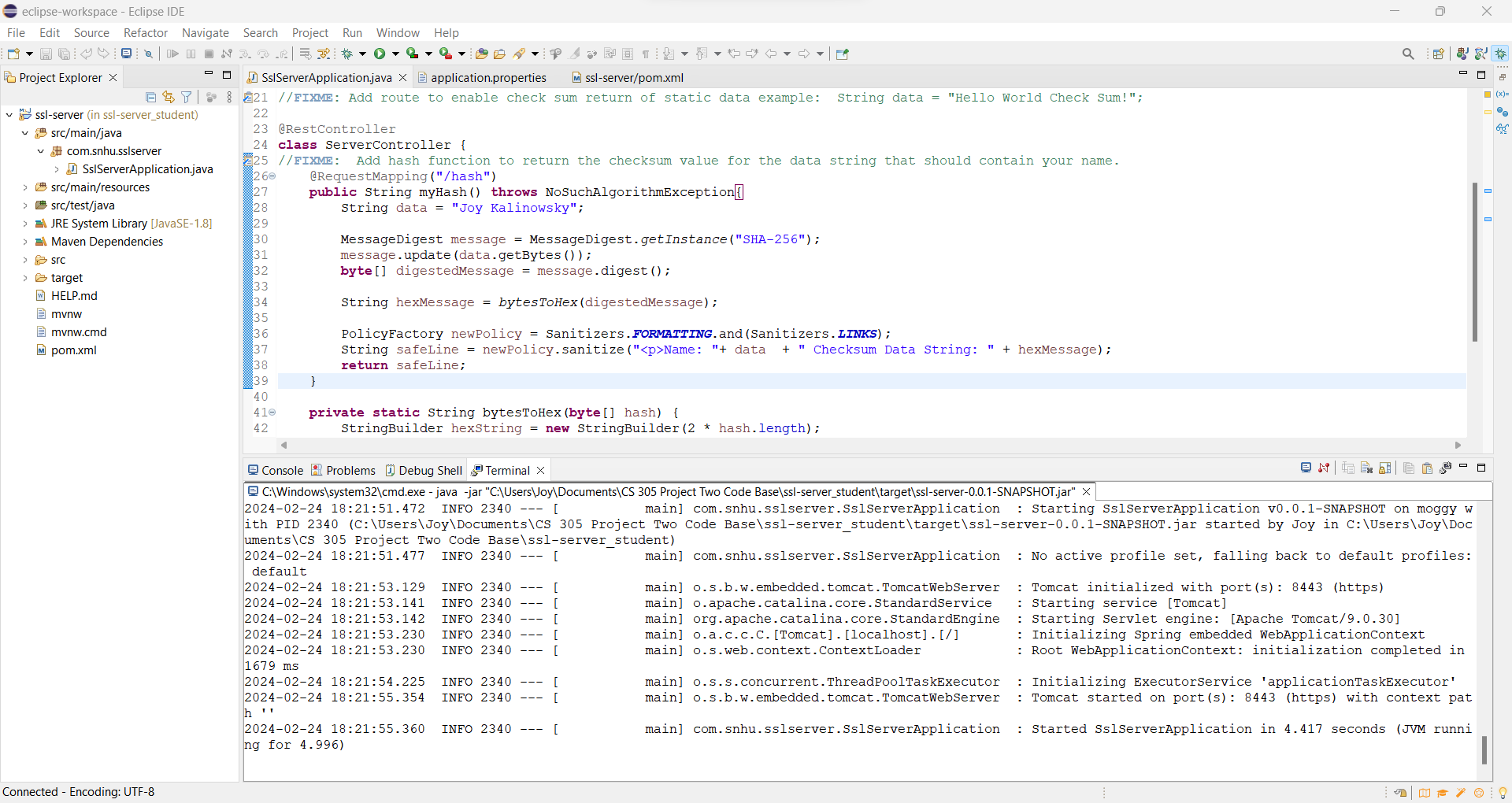
Insert a screenshot below of the web browser that shows a secure webpage.

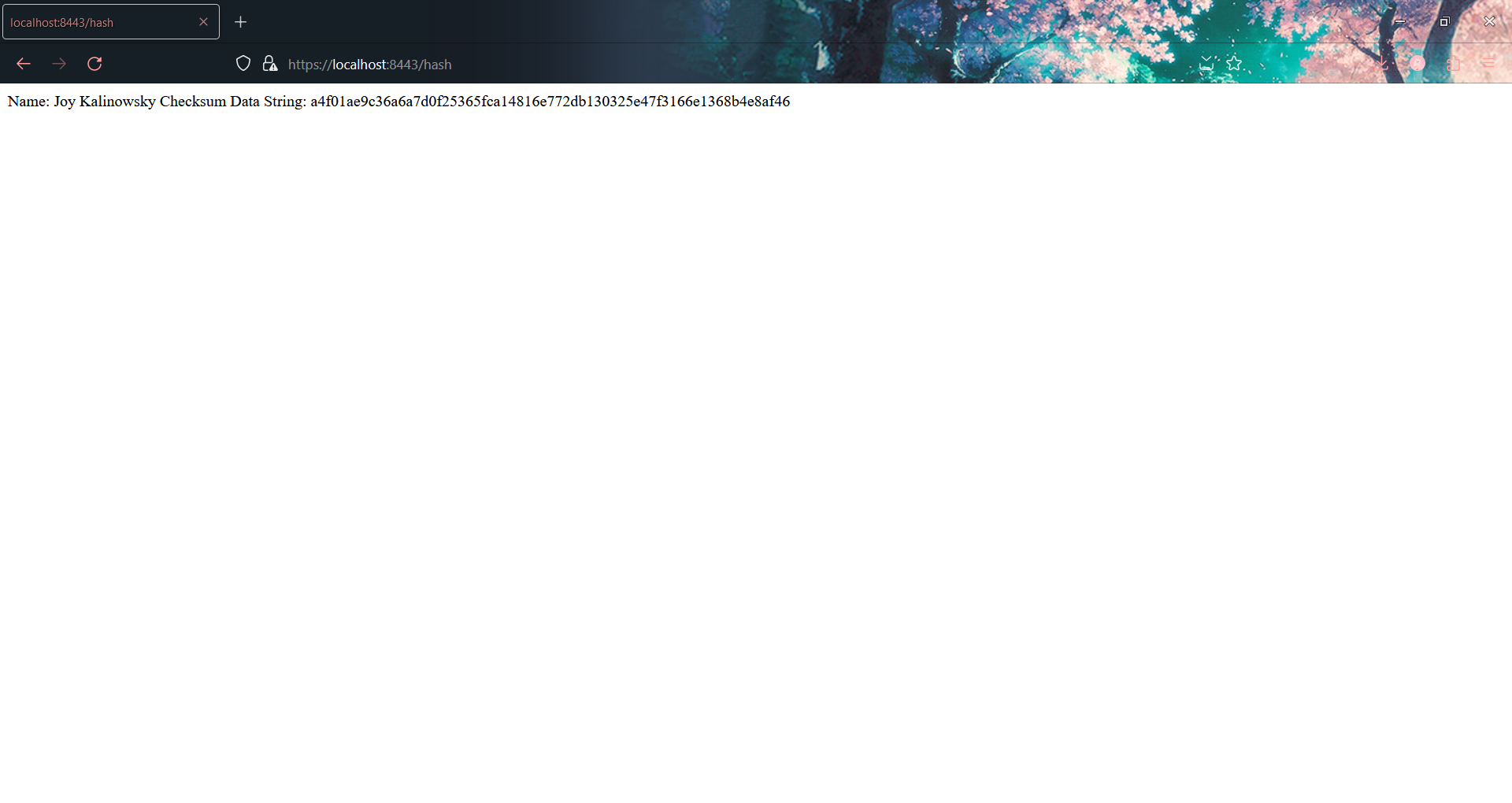


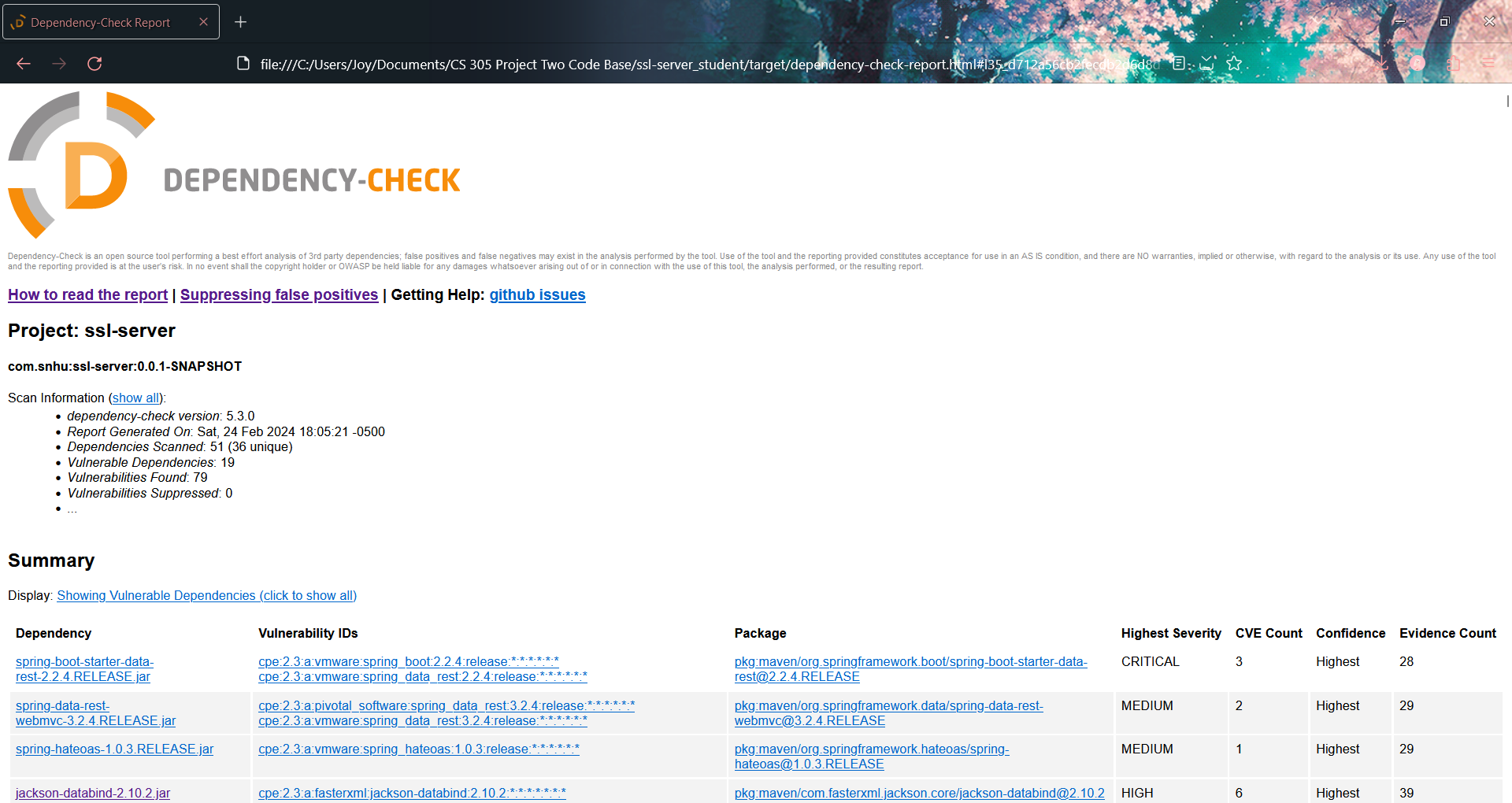
## Secondary Testing

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



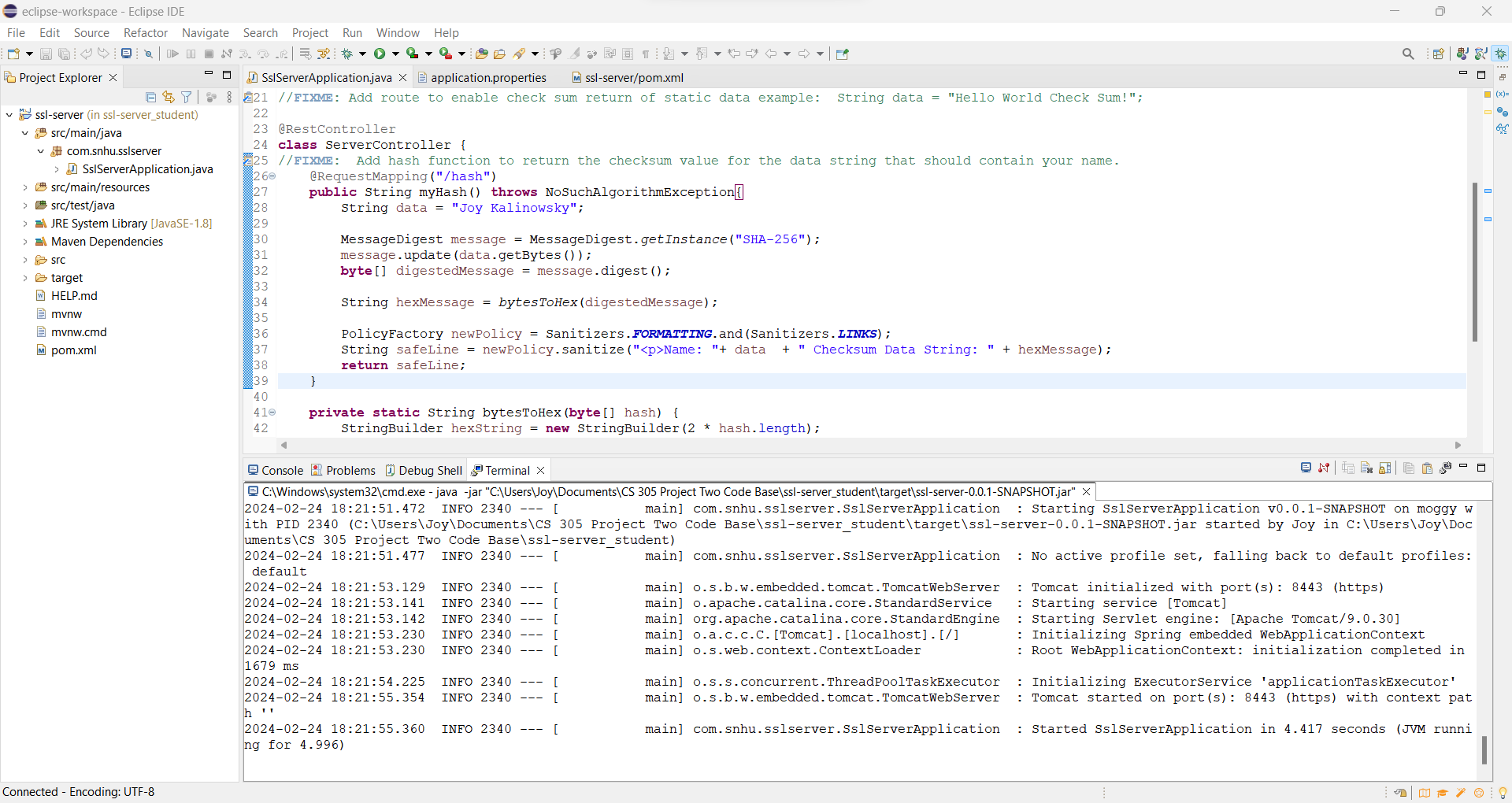






## Functional Testing

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



## Summary

My code has been refactored by converting the old HTTP protocol to the HTTPS protocol, adding an exception handler, and by encoding the output string with the OWASP HTML Sanitizer.

The first way my code has been refactored is by converting the old HTTP protocol to the HTTPS protocol. This was done by modifying the server.port, server.ssl.key-alias, server.ssl.key-store-password, server.ssl-key-store, and server.ssl.key-store-type fields in the application.properties file. This was done to add SSH security to the software application’s hosted website.

The second way my code has been refactored is by adding the exception handler. It handles errors such as NoSuchAlgorithm exceptions to prevent hackers from taking advantage of possible situations where the SHA-256 algorithm cannot be properly used to encrypt the plain text data.

The third way my code has been refactored is by encoding the string that would be posted on the live server website with the OWASP HTML Sanitizer. This is to prevent XSS attacks from compromising my software application website.

The areas of the security in the Vulnerability Assessment Process Flow Diagram that I addressed by refactoring the code are Secure Error Handling and Secure Coding Patterns.

I used Secure Error Handling by making sure no errors can be exploited. I accomplished this with the use of a NoSuchAlgorithm exception, in the case that the SHA-256 algorithm to encode the plain text was unavailable.

I addressed Secure Coding Patterns by importing and using the OWASP HTML sanitizer to prevent XSS attacks from modifying the website text to malicious executable code.

My process for adding layers of security to the software application is to first modify the source code Java, and then to modify the HTTPS protocol. I first refactor the code that determines what the hosted website displays to detect and remove vulnerabilities and then work outwards from there to ensure the security of the processes involved in the software application.

## Industry Standard Best Practices

I used industry best standard practices to maintain the software application’s current security and mitigate against known security vulnerabilities by manually reviewing the vulnerabilities of the software application and making sure that my software application follows OWASP’s secure coding best practices. Since many of the vulnerabilities discovered only apply to earlier versions of the Spring framework, as long as I keep my current version of the Spring framework up to date when developing software, I do not have to worry about these vulnerabilities affecting my software application. Following OWASP’s best practices uses tried and true security concepts to protect against common areas of vulnerability. Applying industry standard best practices for secure coding is also valuable to the company’s overall wellbeing because vulnerabilities in software applications being exploited by hackers can cause major problems for the company. It can cause web-hosted software to go offline, frustrating and driving away customers and even end up leaking the private information of both customers and employees that was hosted on the company software. This can result in millions of dollars lost in terms of broken customer trust, missed business opportunities with customers, compromised company secrets and even intercepted financial transactions. Applying industry standard best practices to company software applications is valuable to the company to help prevent such financially devastating situations from occurring.