****

# Practices for Secure Software Report

**Table of Contents**

[**Document Revision History 3**](#_heading=h.30j0zll)

[**Client 3**](#_heading=h.1fob9te)

[**Instructions 3**](#_heading=h.3znysh7)

[**Developer 4**](#_heading=h.2et92p0)

[**1. Algorithm Cipher 4**](#_heading=h.tyjcwt)

[**2. Certificate Generation 4**](#_heading=h.3dy6vkm)

[**3. Deploy Cipher 4**](#_heading=h.1t3h5sf)

[**4. Secure Communications 4**](#_heading=h.4d34og8)

[**5. Secondary Testing 4**](#_heading=h.2s8eyo1)

[**6. Functional Testing 4**](#_heading=h.17dp8vu)

[**7. Summary 4**](#_heading=h.3rdcrjn)

[**8. Industry Standard Best Practices 4**](#_heading=h.26in1rg)

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **[Date]** | **[Your Name]** |  |

## 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

Alexander Lee Cummings

## Algorithm Cipher

**A. Brief High-Level Overview of the Encryption Algorithm Cipher**Encryption at Artemis Financial involves safeguarding data both in transit (between a client and a server) and at rest (stored on servers or other storage media). For data in transit, asymmetric encryption methods such as RSA are commonly used to initiate a secure handshake. This process typically begins when a user (client) encrypts a request using the server’s public key, ensuring that only the server’s private key can decrypt the message. Once trust is established, a symmetric session key is generated and used for the rest of the communication, as symmetric encryption is faster and more efficient for bulk data transfer. An example of a commonly used cipher suite in this scenario is **TLS\_ECDHE\_RSA\_WITH\_AES\_128\_CBC\_SHA**. For data at rest, symmetric encryption is often preferred—e.g., AES (Advanced Encryption Standard)—because both encryption and decryption are handled by the same system, minimizing the risk of exposing keys during transit.

**B. Hash Functions and Bit Levels of the Cipher**Checksums, generated via hash functions, validate the integrity of data by ensuring that no alterations occur between the time data is sent and received. A one-way hash function such as those in the SHA (Secure Hash Algorithm) family converts the original data into a short, unique, and nearly impossible-to-reverse key. Common hash functions include MD2, MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. However, MD2, MD5, and SHA-1 are considered insecure due to known collisions. Modern recommendations favor SHA-2 algorithms, such as SHA-256 or SHA-512, which offer higher bit rates and drastically reduce the risk of collisions. In this context, the higher the bit rate, the more computationally intensive it is to find a collision, making **SHA-512** particularly robust for high-security requirements

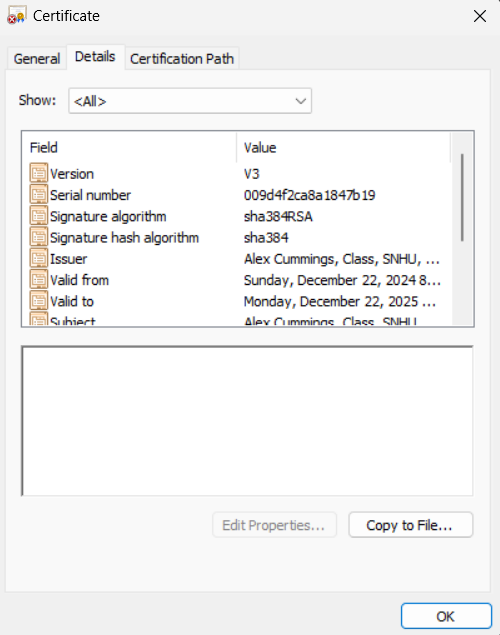
**C. Random Numbers, Symmetric vs. Non-Symmetric Keys, and Other Considerations**A critical component of any cryptographic process is the secure generation of random numbers. Cryptographic keys rely on randomness to prevent predictability. Secure Random Number Generators (RNGs) leverage entropy from system events, timing discrepancies, or even hardware-based modules to produce truly random sequences.

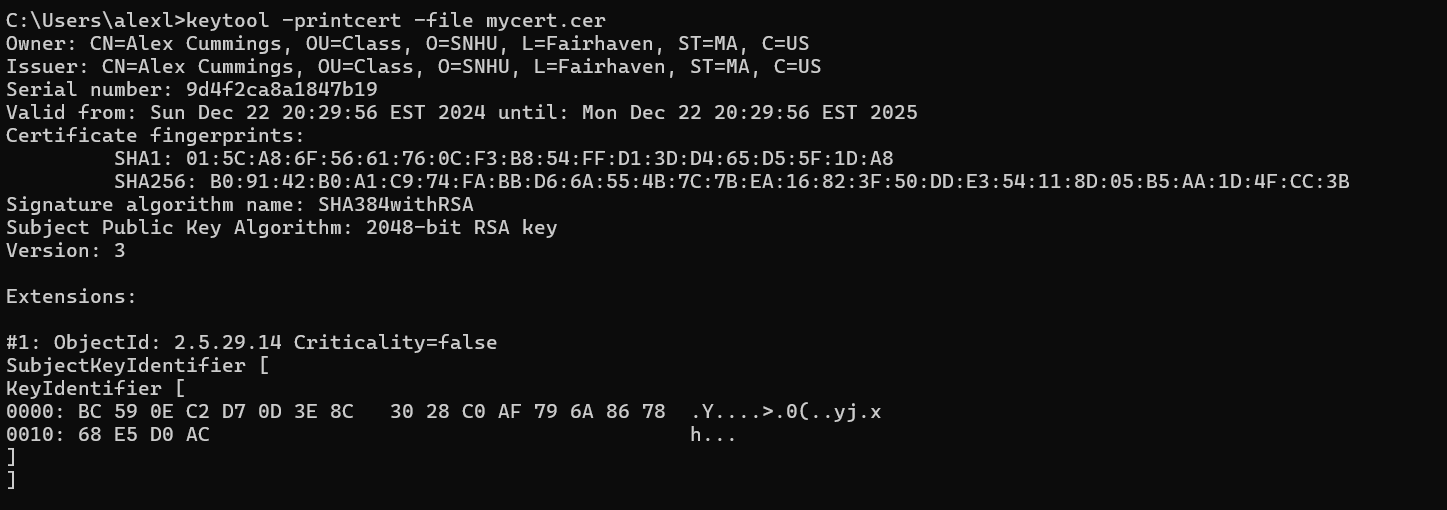
In terms of key usage, **symmetric keys** employ the same key for both encryption and decryption. They are efficient for large volumes of data but require a secure channel for key exchange. **Asymmetric keys**, in contrast, use separate keys for encryption (public key) and decryption (private key). This allows secure exchange of information without sharing private data over the network. In practice, many systems use a **hybrid approach**: asymmetric methods (like RSA) secure the initial key exchange or “handshake,” while symmetric keys (like those used in AES) handle the bulk of data encryption thereafter.

**D. History and Current State of Encryption Algorithms**Historically, ciphers ranged from simple letter substitution methods to more sophisticated polyalphabetic and matrix-based systems used by militaries and governments. These early ciphers relied on shifting letters or replacing them in predictable patterns, which were relatively easy to crack once adversaries recognized linguistic patterns. In modern times, powerful algorithms like **RSA** (Rivest–Shamir–Adleman) and ECC (Elliptic Curve Cryptography) have become standard for secure key exchange and digital signatures, while AES stands out as a robust choice for symmetric encryption.

Looking ahead, additional layers of security such as **two-factor authentication (2FA)** and **multi-factor authentication (MFA)** are increasingly common, combining something the user knows (a password) with something the user has (a phone or token). **Post-quantum cryptography** is also gaining attention, as it aims to develop algorithms capable of resisting attacks from future quantum computers. Thus, cryptography continues to evolve to meet new threats and computational advances, ensuring data remains protected across diverse communication channels and storage solutions.

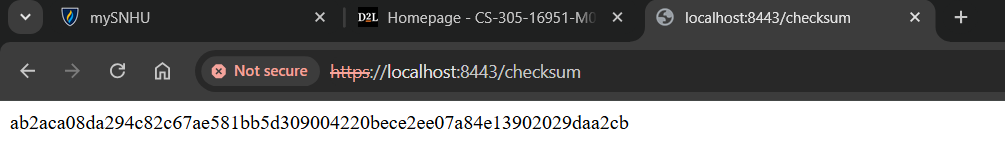
## Certificate Generation

Insert a screenshot below of the CER file.



## Deploy Cipher

Insert a screenshot below of the checksum verification.



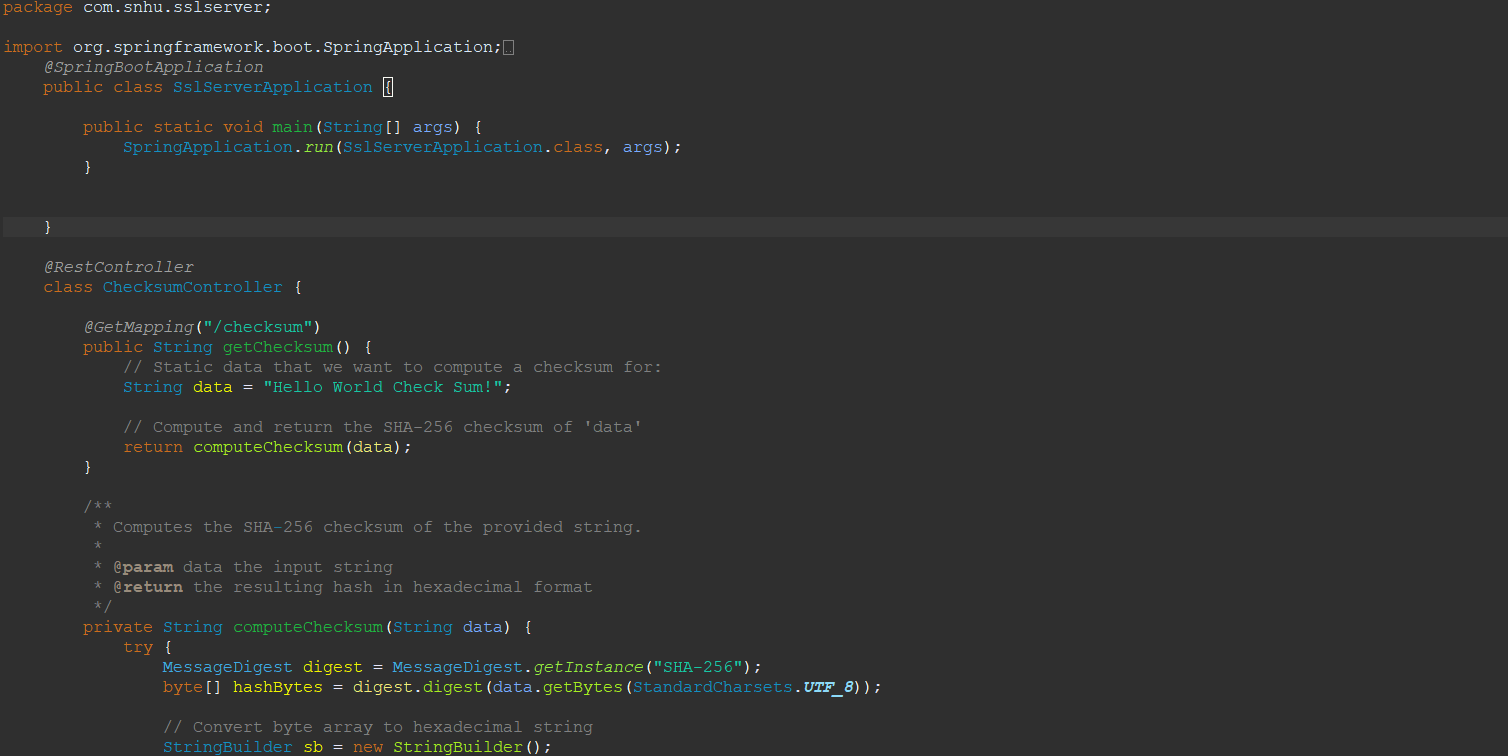
## Secure Communications

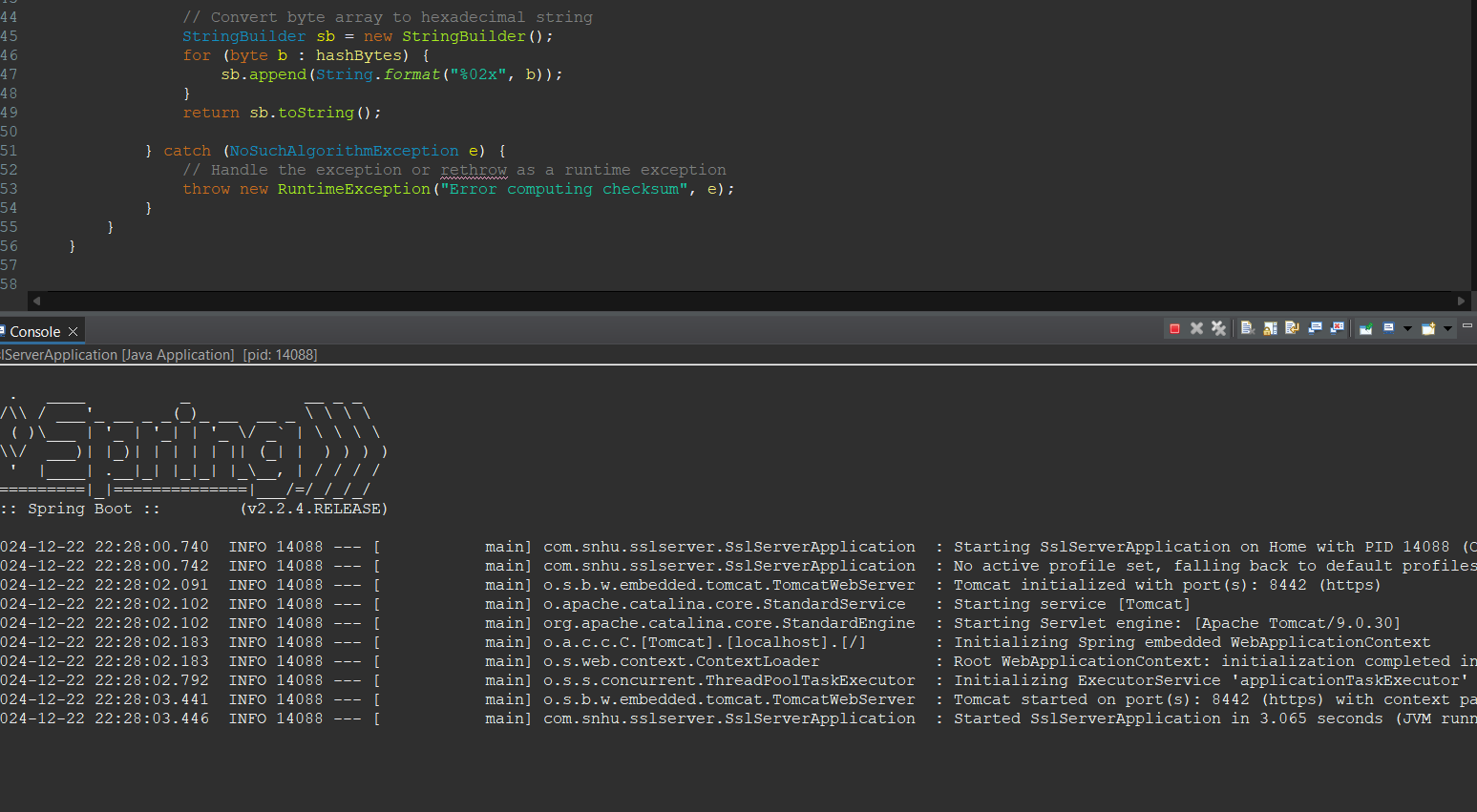
Insert a screenshot below of the web browser that shows a secure webpage.

Reference screenshot above, I am already using the HTTPS protocol

## Secondary Testing

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

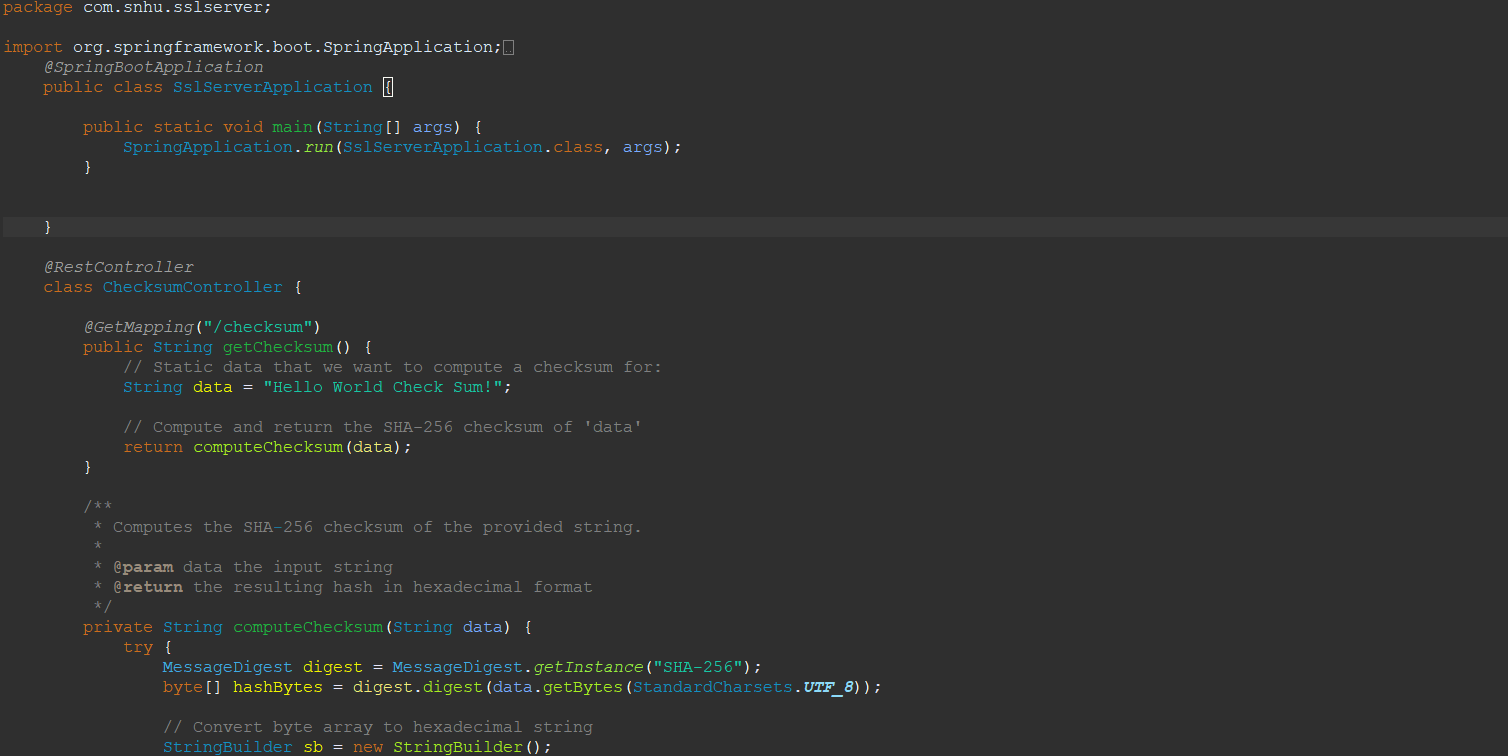


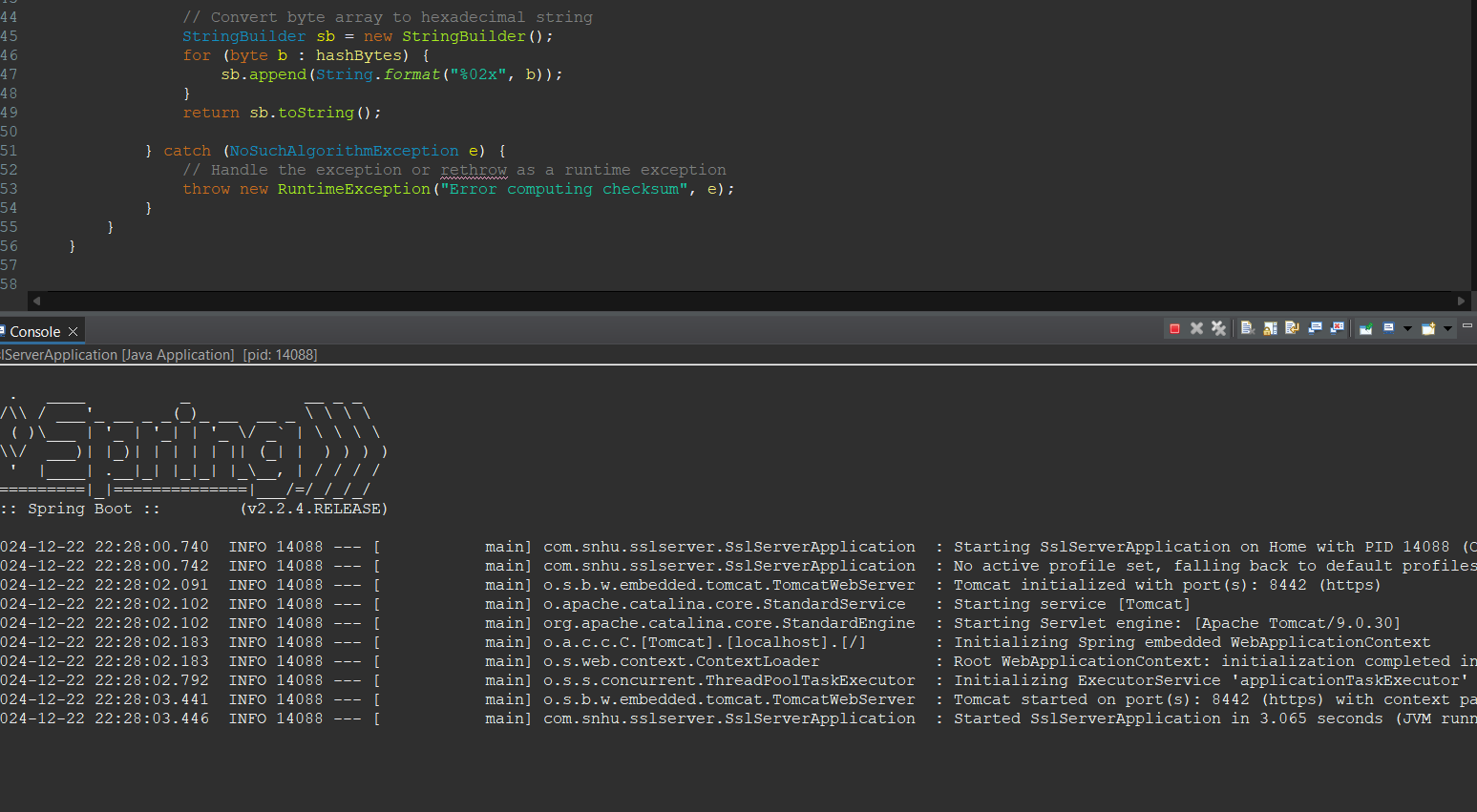


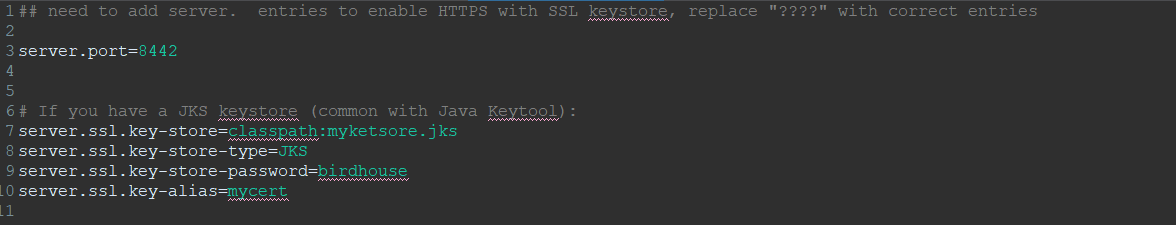
Above screenshots show refactored code running properly with no errors as demonstrated by the console message in the second image.

## Functional Testing

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







I conducted a manual review of the refactored code focusing on syntactical, logical, and security considerations. During this review, I ensured there were no unused imports, syntactic errors, or unhandled exceptions. I confirmed that the checksum functionality works as intended without exposing sensitive data, and that any new methods follow secure practices (e.g., using up-to-date hashing algorithms). Finally, I ran the application, observed a clean startup in the console, and verified the checksum endpoint returns the expected SHA-256 hash. No additional vulnerabilities were introduced by these code changes.

## Summary

In addressing vulnerabilities found through the vulnerability assessment process, I focused on two primary areas: data integrity and transport security. First, I reviewed the original code to identify any outdated or weak hashing methods that could expose user information to collision attacks. Replacing such methods with SHA-256 provided a stronger, more collision-resistant algorithm. Additionally, I ensured the application utilized valid SSL configurations and certificates, reducing the risk of man-in-the-middle attacks while data is in transit. This approach included generating a properly secured keystore and referencing it within the Spring Boot settings to establish encrypted communications over HTTPS.

Next, I tried to perform a static analysis using OWASP Dependency-Check to confirm that no newly introduced libraries contained known security risks. Again, I was unable to get Maven to run properly through Eclipse, so I cannot confirm that the introduced libraries contained any known security risks. I also conducted a manual code review to verify that none of the changes introduced logical flaws or unsecured configurations. Through these measures, I established multiple layers of defense: robust hashing for integrity, TLS for secure transport, and ongoing dependency checks for emerging vulnerabilities. Ultimately, these enhancements help ensure that the refactored code aligns with modern security best practices without compromising performance or functionality.

## Industry Standard Best Practices

Adhering to industry-standard best practices ensures that the organization’s applications are resilient against common attack vectors, which safeguards both customer data and the company’s reputation. Implementing robust encryption protocols, regularly scanning for dependencies with known vulnerabilities, and following secure coding guidelines mitigate the chances of a costly data breach. In addition, such practices foster customer trust and confidence, as clients and end-users are more likely to engage with a company that demonstrably values data integrity and confidentiality.

From a business perspective, secure coding principles not only reduce the risk of legal liability and regulatory penalties but also help maintain operational continuity. Security incidents can lead to downtime, loss of revenue, and damage to the brand, whereas proactive compliance with best practices lowers these risks significantly. Ultimately, a strong security posture enables the organization to innovate freely, build client trust, and stay ahead of evolving threats—ensuring long-term sustainability and a competitive edge in the marketplace.