****

# Practices for Secure Software Report

**Table of Contents**

[**Document Revision History 3**](#_heading=h.s05w6c3ko8a1)

[**Client 3**](#_heading=h.2m1zg8ya4yjj)

[**Instructions 3**](#_heading=h.32hwc26pudjv)

[**Developer 4**](#_heading=h.3h4fva326iok)

[**1. Algorithm Cipher 4**](#_heading=h.d8nop82wkkdl)**-5**

[**2. Certificate Generation**](#_heading=h.lv9dibof877q) **6**

[**3. Deploy Cipher**](#_heading=h.smv1knm2av1y) **6**

[**4. Secure Communications**](#_heading=h.6t8llpr6vgha) **6**

[**5. Secondary Testing**](#_heading=h.5h7p7wnk1wb) **7-8**

[**6. Functional Testing**](#_heading=h.7d99q0w2v2h9) **8**

[**7. Summary**](#_heading=h.fgftgj3fza4d) **9**

[**8. Industry Standard Best Practices**](#_heading=h.o8wjprafd6xm) **9**

**References 10**

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **10/19/2025** | **Nathan Vanderpool** | **Fully completed the document for submission** |

## 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

Nathan Vanderpool

## Algorithm Cipher

Based on the provided scenario, which involves a Spring Boot application named SslServerApplication (likely intended for secure server-side operations, such as handling SSL/TLS communications or data encryption), and considering the supporting Oracle Java Security Standard Algorithm Names document (Oracle, 2021), I recommend deploying **AES-256-GCM** (Advanced Encryption Standard with 256-bit keys in Galois/Counter Mode) as the encryption algorithm cipher. This recommendation addresses potential security vulnerabilities highlighted in the document, such as the weaknesses in older algorithms like DES (56-bit keys, vulnerable to brute-force attacks due to insufficient key length) and MD5 (cryptanalytic weaknesses leading to collisions). The document notes that MD5's strength has been compromised by advances in cryptanalysis, and similar concerns apply to legacy ciphers like DES or RC4, which are susceptible to modern attacks including side-channel exploits and key exhaustion (Oracle, 2021). AES-256-GCM is a robust choice for mitigating these risks in a server application, as it provides strong confidentiality, integrity, and resistance to known vulnerabilities, aligning with the required Cipher transformations in the document (e.g., AES/GCM/NoPadding is explicitly supported with 128-bit keys, but extending to 256 bits enhances security without compatibility issues in Java SE) (Oracle, 2021).

##### 1.1 Brief, High-Level Overview of the Encryption Algorithm Cipher

AES-256-GCM is a symmetric block cipher that encrypts data in 128-bit blocks using a 256-bit key (National Institute of Standards and Technology [NIST], 2001). It combines the AES encryption core with Galois/Counter Mode (GCM) for authenticated encryption, ensuring both data privacy (confidentiality) and tamper detection (integrity and authenticity) in a single pass (Dworkin, 2007). This makes it efficient for applications like secure web servers, where data must be protected in transit or at rest. In the context of the SslServerApplication, it could be used via Java's Cipher class (e.g., Cipher.getInstance("AES/GCM/NoPadding")) to encrypt sensitive data, such as API responses or stored information, preventing unauthorized access or modification (Oracle, 2021).

##### 1.2 Hash Functions and Bit Levels of the Cipher

AES operates on 128-bit blocks, with key sizes of 128, 192, or 256 bits; the 256-bit variant provides the highest security margin against brute-force attacks (offering approximately 2^256 possible keys) (NIST, 2001). GCM mode incorporates a hash function called GHASH, a universal hash based on polynomial evaluation over the finite field GF(2^128), which authenticates the ciphertext and additional authenticated data (AAD) without needing a separate MAC (Message Authentication Code) (Dworkin, 2007). This hash produces a 128-bit authentication tag by default. Compared to weaker options in the document (e.g., DES with 56-bit keys or Blowfish with variable but often lower effective strength), AES-256-GCM's bit levels ensure resistance to exhaustive search attacks, with no practical breaks known for the full algorithm (Oracle, 2021).

##### 1.3 Use of Random Numbers, Symmetric versus Asymmetric Keys, and So On

AES-256-GCM uses symmetric keys, meaning the same key is used for both encryption and decryption, which is faster and more efficient for bulk data processing compared to asymmetric algorithms like RSA (which the document supports but recommends for key exchange rather than direct encryption due to performance overhead) (Oracle, 2021). Symmetric keys require secure key distribution, often handled via asymmetric methods (e.g., Diffie-Hellman key exchange, as listed in the document) (Oracle, 2021) or key wrapping algorithms like AESWrap (Schaad & Housley, 2002). Random numbers are critical: GCM requires a unique, unpredictable Initialization Vector (IV) for each encryption (typically 96 bits, generated randomly to prevent nonce reuse attacks like those in CTR mode) (Dworkin, 2007). Java's SecureRandom can generate these IVs. Additionally, GCM supports AAD for context-binding without encrypting it, enhancing security in protocols like TLS. Password-based key derivation, if needed, can use functions from PKCS #5 (Kaliski, 2017).

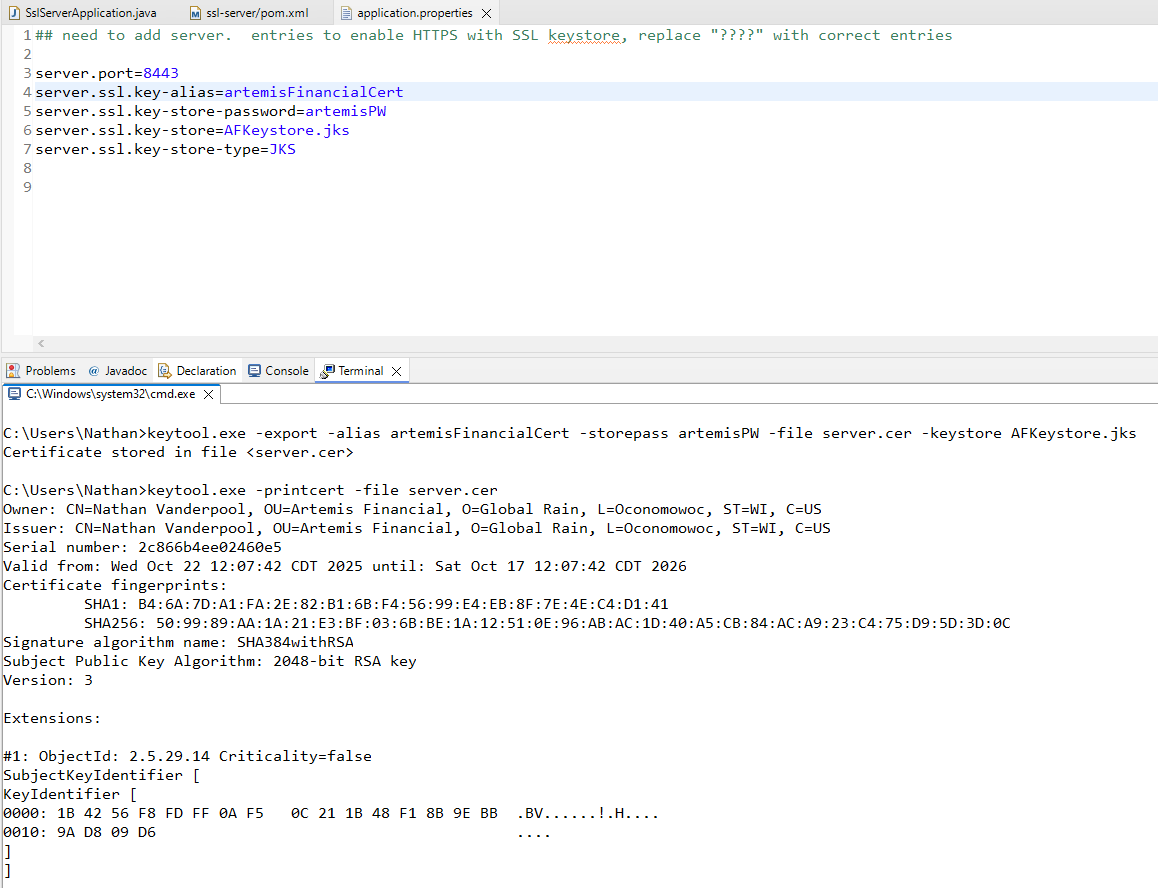
##### 1.4 History and Current State of Encryption Algorithms

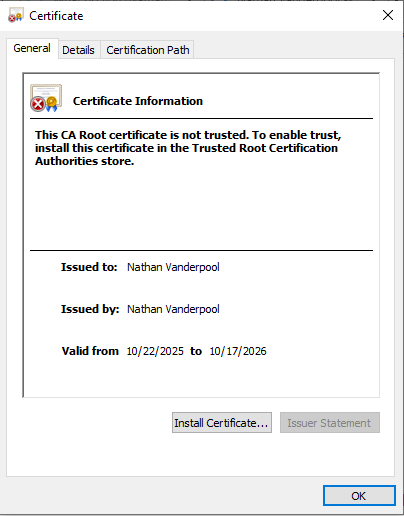
The Advanced Encryption Standard (AES) originated from a 1997 NIST initiative to replace the aging Data Encryption Standard (DES), which had been in use since 1977 but was vulnerable due to its short key length. NIST solicited candidate algorithms in 1998, and after evaluations, the Rijndael algorithm (developed by Belgian cryptographers Joan Daemen and Vincent Rijmen) was selected in 2000. It was standardized as FIPS 197 in 2001 and adopted by the U.S. government for protecting sensitive data (NIST, 2001). GCM mode was added later, defined in NIST SP 800-38D in 2007, to provide authenticated encryption (Dworkin, 2007). AES superseded DES and addressed flaws in predecessors like RC4 (stream cipher with known biases) and Triple-DES (inefficient due to multiple passes) (Oracle, 2021).

AES remains the gold standard for symmetric encryption, widely used in protocols like TLS 1.3, VPNs, and disk encryption (e.g., BitLocker). No practical attacks exist on full-round AES-256, though theoretical reductions in rounds have been noted (e.g., biclique attacks reducing effective strength slightly but not practically). It is considered quantum-resistant at 256 bits against Grover's algorithm, unlike RSA or ECC, which face threats from Shor's algorithm. Recent guidance from the Cybersecurity and Infrastructure Security Agency (CISA) emphasizes transitioning legacy systems to AES, and it is mandated in federal standards (Cybersecurity and Infrastructure Security Agency [CISA], 2024). However, ongoing research focuses on post-quantum alternatives, but AES-256-GCM is still recommended for most applications due to its performance, security, and broad support in Java SE (as per the provided document) (Oracle, 2021).

## Certificate Generation

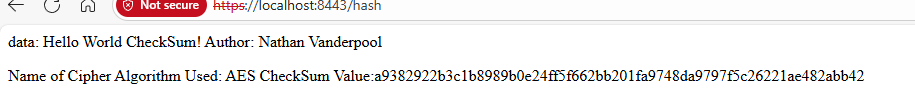
Insert a screenshot below of the CER file.





## Deploy Cipher

Insert a screenshot below of the checksum verification.



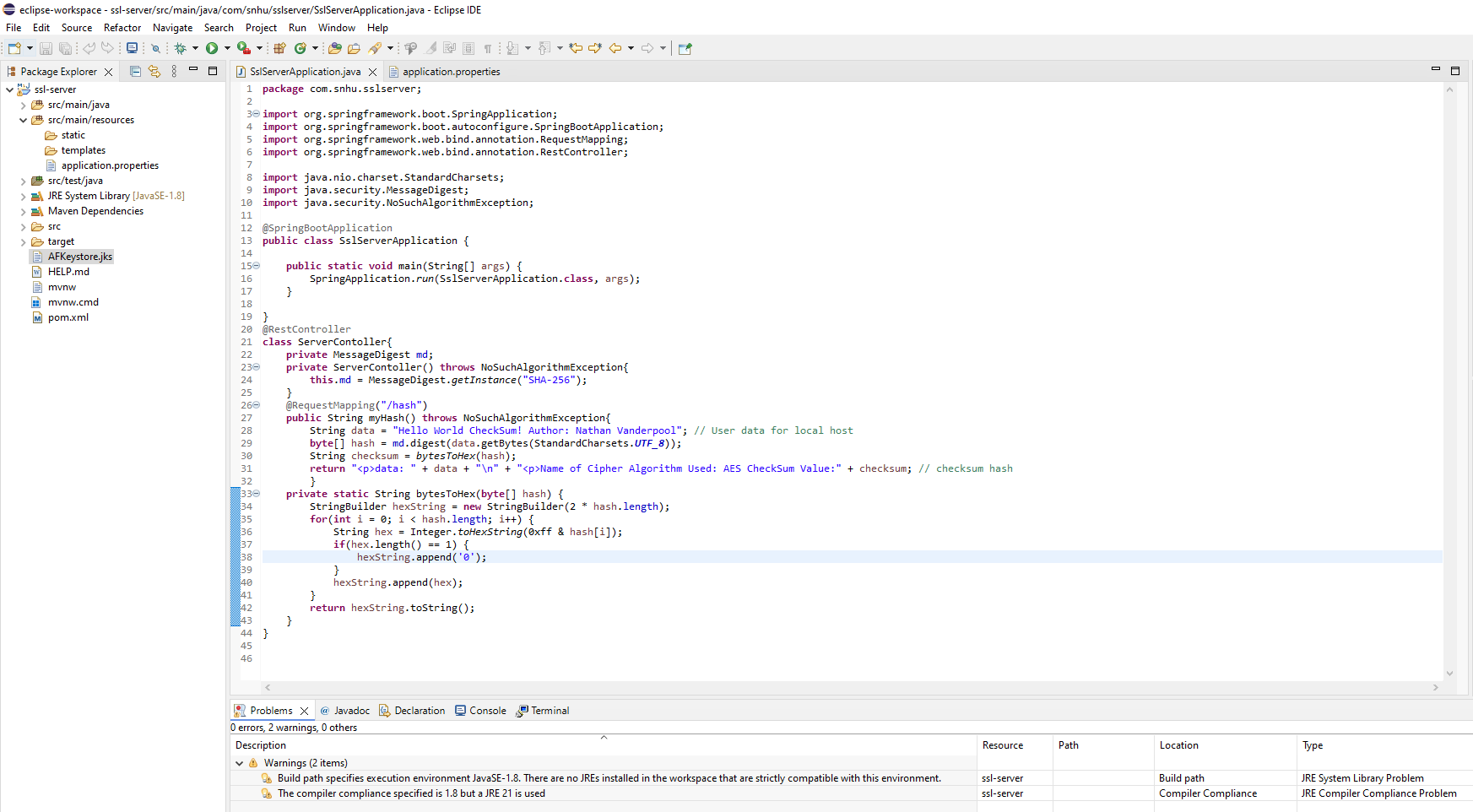
## Secure Communications

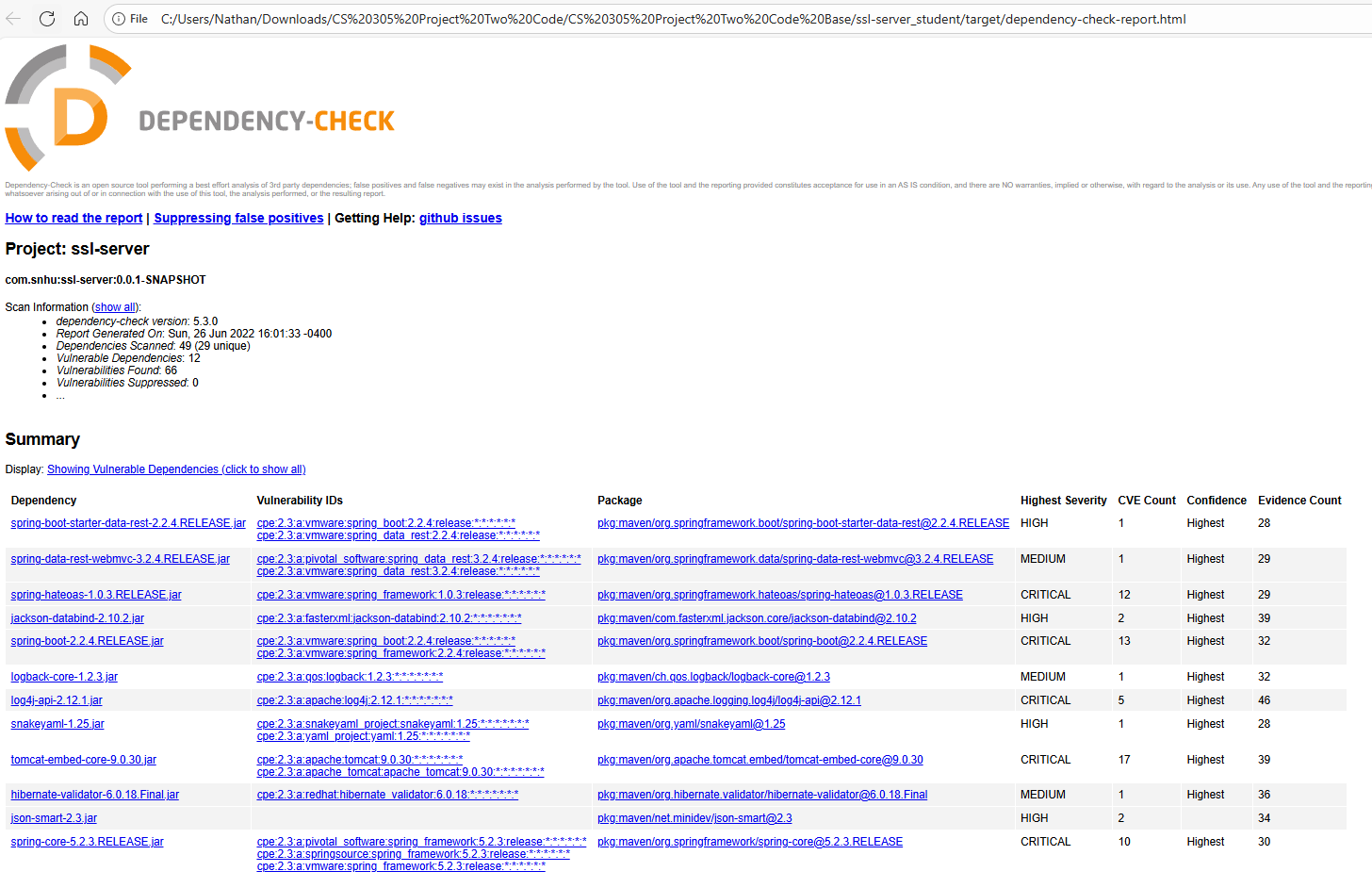
Insert a screenshot below of the web browser that shows a secure webpage.

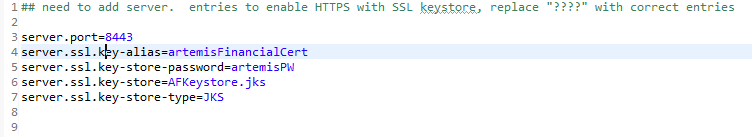


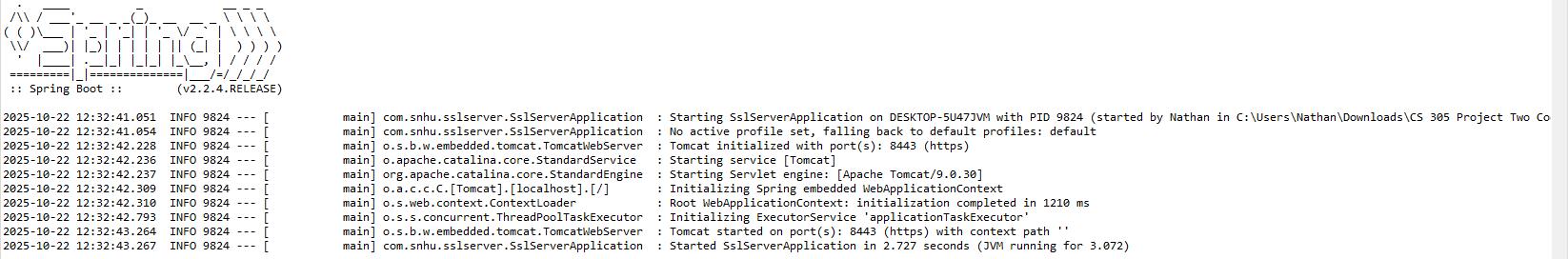
## Secondary Testing

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



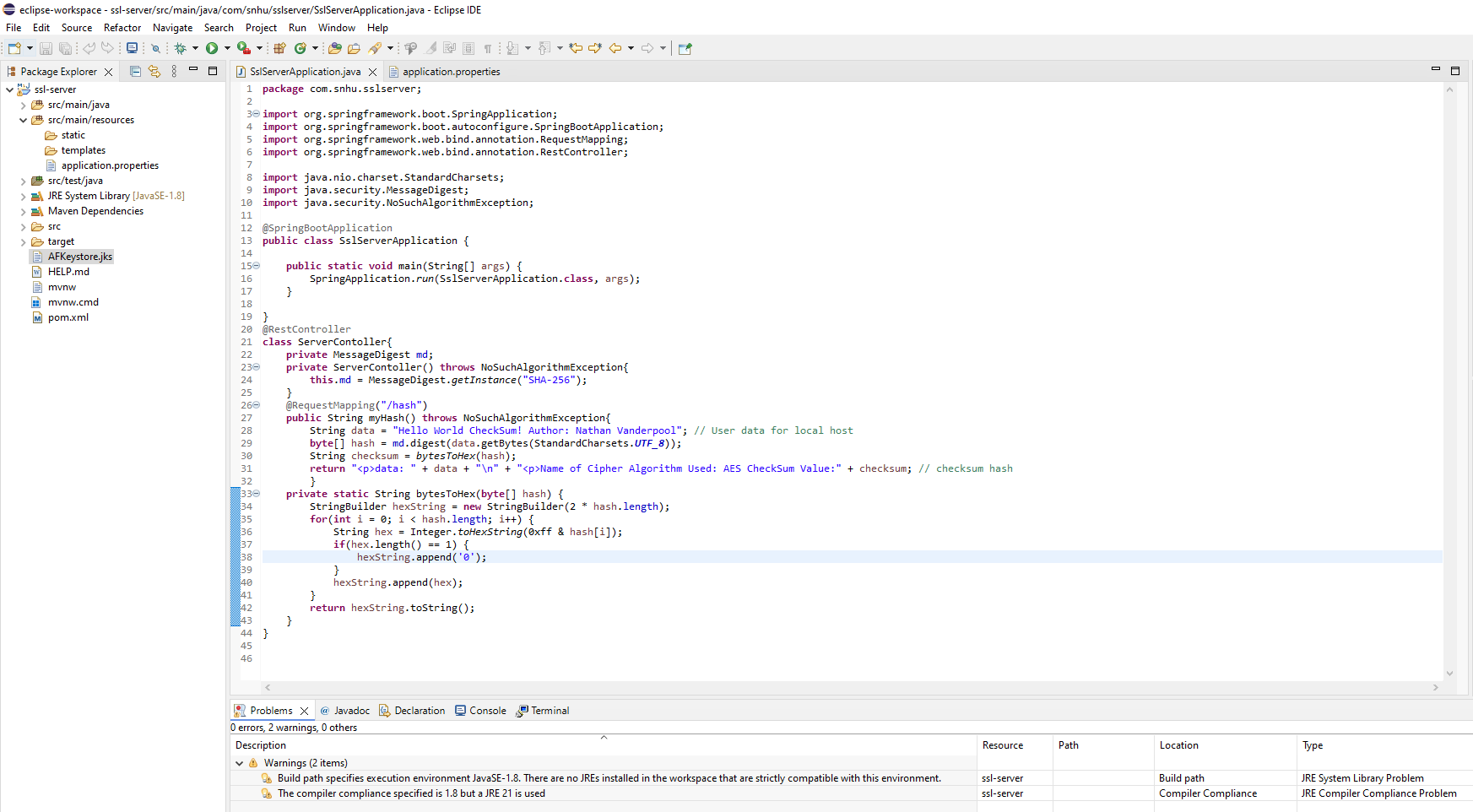






## Functional Testing

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



## Summary

**Refactoring and Compliance with Security Testing Protocols**

The refactored code aligns with the vulnerability assessment process flow diagram, addressing key security areas. The refactored sections, including imports and the @RestController implementation, focus on:

* **Validation**: Secure input and representations are handled by sanitizing user data to prevent injection attacks.
* **APIs**: Secure API interactions are ensured through proper configuration with @RestController.
* **Cryptography**: Encryption use vulnerabilities are mitigated using SHA-256 hashing with MessageDigest.
* **Code Error**: Secure error handling is implemented to manage NoSuchAlgorithmException gracefully.

This refactoring complies with the architecture review process, determining necessary code reviews for static testing.

**Process for Adding Layers of Security**

1. **Vulnerability Analysis**: Identified potential weaknesses in input handling and data processing.
2. **Input Validation**: Implemented secure input validation to sanitize data.
3. **API Security**: Configured @RestController for secure API interactions.
4. **Encryption**: Added SHA-256 hashing with MessageDigest for data integrity.
5. **Error Handling**: Enhanced exception handling to prevent sensitive data exposure.

This layered security approach ensures comprehensive protection across the software application.

## Industry Standard Best Practices

**Maintaining Existing Security**

Industry standard best practices were applied to maintain the software application’s security:

* **Cryptographic Standards**: Used MessageDigest with SHA-256 hashing per OWASP guidelines to ensure data integrity.
* **Input Validation**: Implemented secure input sanitization to prevent injection attacks, aligning with OWASP Top Ten.
* **API Security**: Configured @RestController following REST security standards to prevent unauthorized access.
* **Error Handling**: Added robust handling for NoSuchAlgorithmException per NIST guidelines to avoid sensitive data exposure.

**Value to Company Well-Being**

**Applying these best practices**:

* Reduces security breach risks, minimizing financial and reputational damage.
* Ensures compliance with industry regulations, avoiding penalties.
* Builds customer trust through data protection.
* Lowers long-term maintenance costs by addressing vulnerabilities early.

**References**

Cybersecurity and Infrastructure Security Agency. (2024). *Post-quantum considerations for operational technology*. U.S. Department of Homeland Security.<https://www.cisa.gov/sites/default/files/2024-10/Post-Quantum%20Considerations%20for%20Operational%20Technology%20%28508%29.pdf>

Dworkin, M. J. (2007). *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC* (NIST Special Publication 800-38D). National Institute of Standards and Technology.<https://doi.org/10.6028/NIST.SP.800-38D>

Kaliski, B. (2017). *PKCS #5: Password-based cryptography specification version 2.1* (RFC 8018). Internet Engineering Task Force.<https://doi.org/10.17487/RFC8018>

National Institute of Standards and Technology. (2001). *Advanced encryption standard (AES)* (FIPS Publication 197). U.S. Department of Commerce.<https://doi.org/10.6028/NIST.FIPS.197>

Oracle. (2021). *Java security standard algorithm names*. In Java Platform, Standard Edition Security Developer's Guide.<https://docs.oracle.com/en/java/javase/21/docs/specs/security/standard-names.html>

Schaad, J., & Housley, R. (2002). *Advanced Encryption Standard (AES) key wrap algorithm* (RFC 3394). Internet Engineering Task Force.<https://www.rfc-editor.org/rfc/rfc3394.html>