Department of Electronic and Telecommunication Engineering

University of Moratuwa

200123H	DHANOMIKA A.P.N.
200134R	DISSANAYAKA R.M.R.H.
200310E	KODITHUWAKKU K.A.W.T.
200702H	WEERASINGHE P.D.G.U.M.B.

Team CyberSphere

EN4720: Security in Cyber-Physical Systems

Course Project

The design and functionality of the cryptographic APIs.

Key Generation

Functionality:

- Generates AES or RSA keys based on user specifications.
- Stores the generated keys in an in-memory dictionary for easy retrieval.
- Returns the generated key ID and the key value (for AES) or public key (for RSA).

Supported Key Types:

- AES: Supports key sizes of 128, 192, and 256 bits.
- RSA: Supports variable key sizes for public-private key pairs (2048, 3072, 4096).

Implementation:

- AES keys are generated using `os.urandom`.
- RSA keys are created using `cryptography.hazmat.primitives.asymmetric.rsa.generate_private_key`.

Encryption

Functionality:

- Encrypts plaintext using AES (symmetric) or RSA (asymmetric) encryption.
- AES encryption uses CBC mode with PKCS7 padding.
- RSA encryption uses OAEP padding with SHA-256.

Implementation:

- For AES:
  - Retrieves the stored AES key.
  - Uses an initialization vector (IV) for CBC mode.
  - Pads the plaintext to fit AES block size.
  - Encrypts the data and returns a base64-encoded string.
- For RSA:
  - Uses the stored RSA public key to encrypt the plaintext.
  - Applies OAEP padding for security.
  - Returns the base64-encoded ciphertext.

Decryption

Functionality:

- Decrypts ciphertext encrypted with AES or RSA.
- Removes padding after decryption for AES.

Implementation:

- For AES:
    - Retrieves the stored AES key.
    - Extracts the IV from the received ciphertext.
    - Decrypts the data and removes padding.
- For RSA:
    - Uses the stored RSA private key to decrypt the ciphertext.
    - Applies the same OAEP padding scheme used during encryption.

Hashing

Functionality:

- Computes a cryptographic hash of the input data.
- Supports multiple hashing algorithms.

Supported Algorithms:

- SHA-256, SHA-512, SHA-1, MD5, SHA-224, SHA-384, BLAKE2b, BLAKE2s

Implementation:

- Uses Python's `hashlib` library to compute the hash.
- Returns the base64-encoded hash value.

Hash Verification

Functionality:

- Verifies whether a given hash matches the computed hash of the provided data.
- Returns a boolean indicating the validity of the hash.

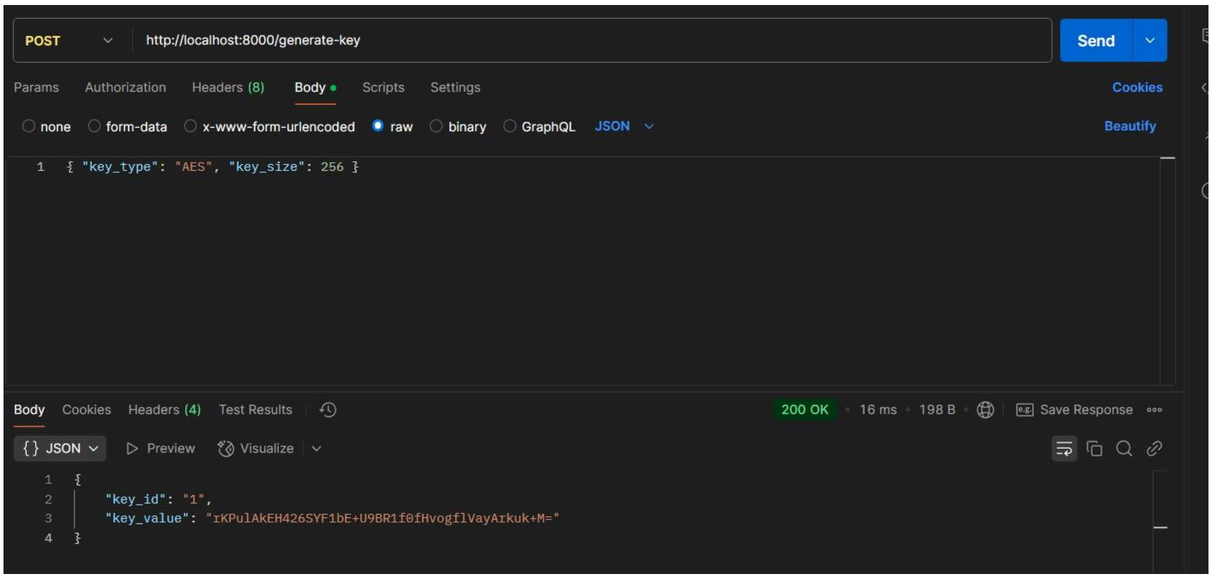
Implementation:

- Computes the hash using the same algorithm.
- Compare it with the provided hash.

Screenshots of successful API operations (encryption/decryption & hashing demonstrations).

Symmetric key encryption , decryption

❖ We provide an algorithm and a key size, then we get a key value for that.



POST http://localhost:8000/generate-key

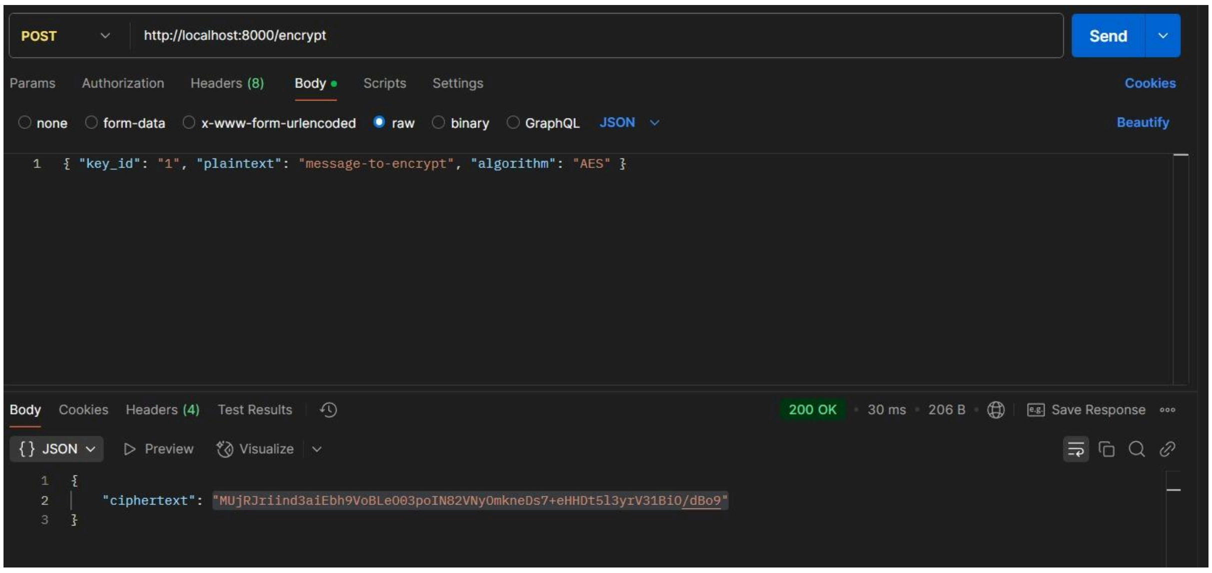
Body: raw JSON

```
{ "key_type": "AES", "key_size": 256 }
```

200 OK · 16 ms · 198 B

```
{
    "key_id": "1",
    "key_value": "rKPulAkEH426SYF1bE+U9BR1f0fHvogflVayArkuk+M="
}
```

❖ At this stage, we provide a key ID, algorithm, and message. Our message is: 'message to encrypt.' Then, we obtain a ciphertext.



POST http://localhost:8000/encrypt

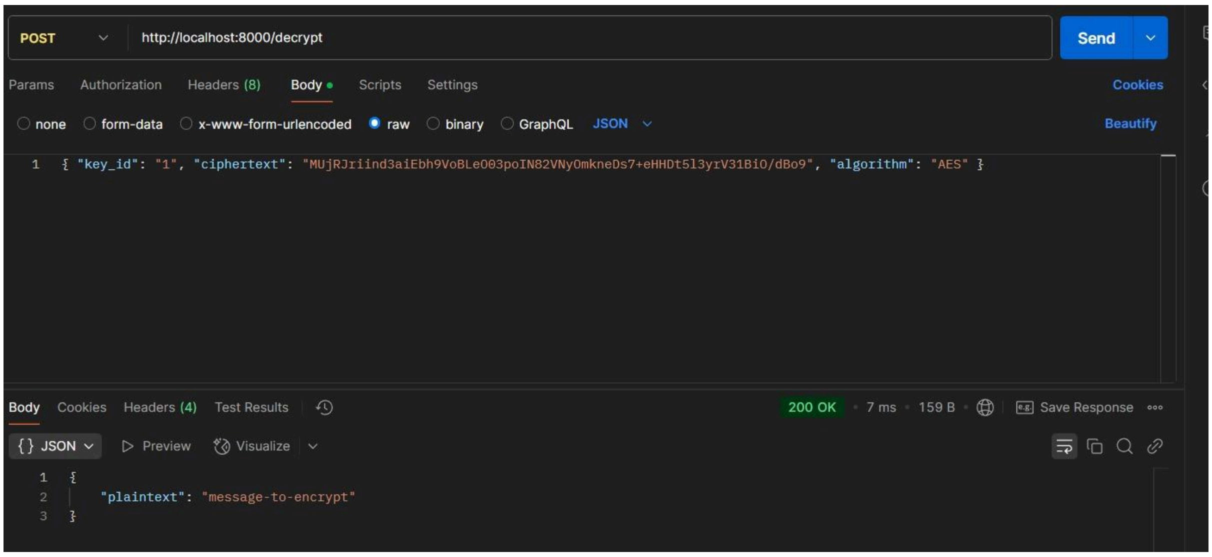
Body: raw JSON

```
{ "key_id": "1", "plaintext": "message-to-encrypt", "algorithm": "AES" }
```

200 OK · 30 ms · 206 B

```
{
    "ciphertext": "MUjRJriind3aiEbh9VoBLeO03poIN82VNyOmkneDs7+eHHDt5l3yrV31BiO/dBo9"
}
```

❖ In this step, we provide the key ID and algorithm along with the ciphertext. This decrypts the ciphertext, and we receive the plaintext. Now, we can see that the message we sent has been received


POST http://localhost:8000/decrypt

Body: raw JSON

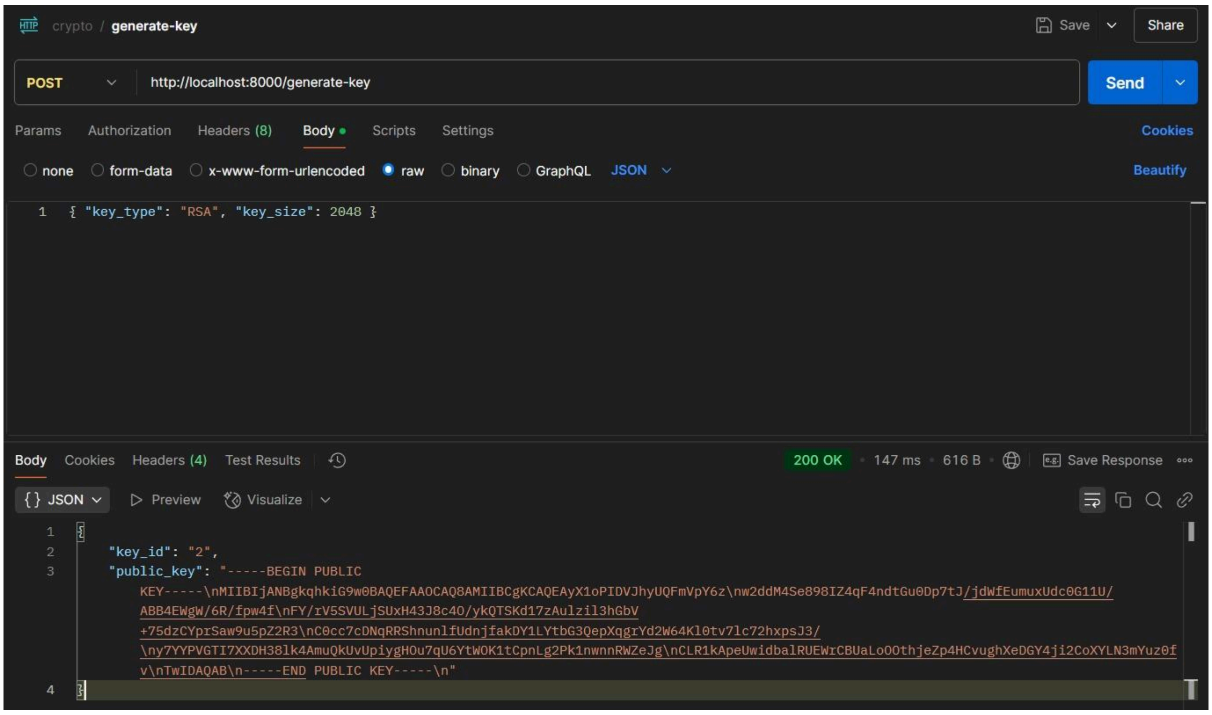
```
{ "key_id": "1", "ciphertext": "MUjRJriind3aiEbh9VoBLeO03poIN82VNyOmkneDs7+eHHDt5l3yrV31BiO/dBo9", "algorithm": "AES" }
```

200 OK · 7 ms · 159 B

```
{
    "plaintext": "message-to-encrypt"
}
```

Asymmetric key encryption , decryption

❖ We provide the key type and key size, then we receive the key ID and public key


crypto / generate-key

POST http://localhost:8000/generate-key

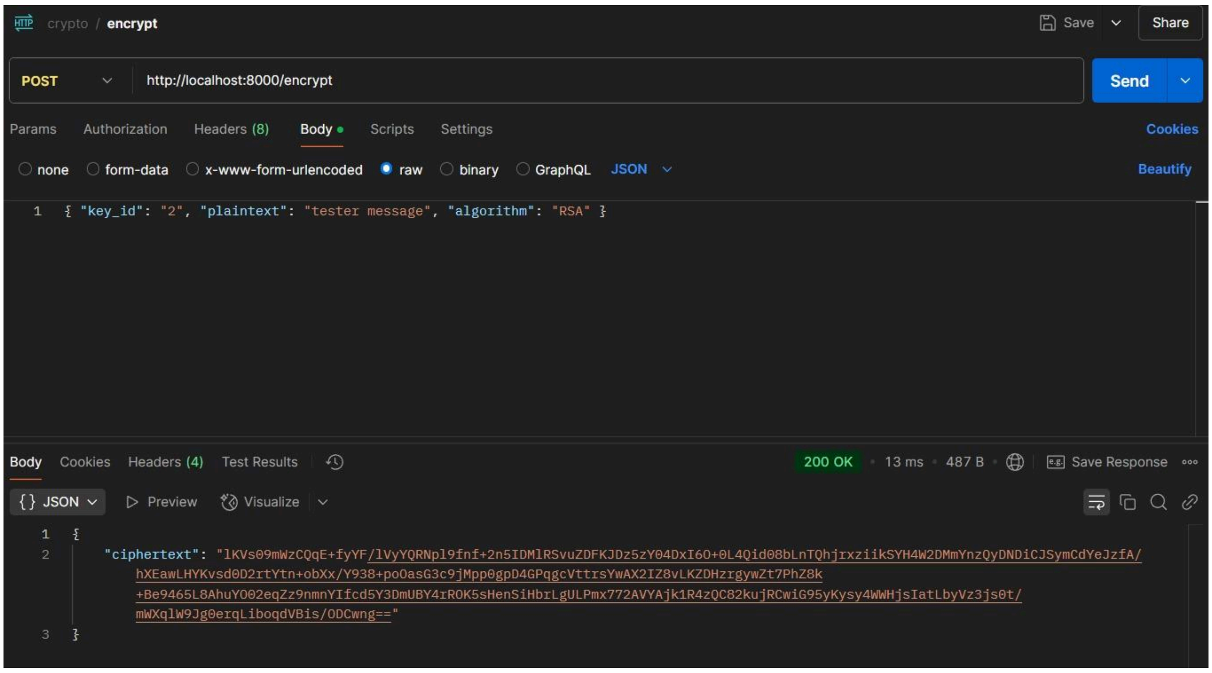
Body: raw JSON

```
{ "key_type": "RSA", "key_size": 2048 }
```

200 OK · 147 ms · 616 B

```
{
    "key_id": "2",
    "public_key": "-----BEGIN PUBLIC
        KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAyX1oPIDVJhyUQFmVpY6z\nw2ddM4Se898IZ4qF4ndtGu0Dp7tJ/jdWfEumuxUdc0G11U/
        ABB4EWgW/6R/fpw4f\nFY/rV5SVULjSUxH43J8c40/ykQTSKd17zAulzil3hGbV
        +75dzCYprSaw9u5pZ2R3\nC0cc7cDNqRRShnunlfUdnjfakDY1LYtbG3QepXqgrYd2W64Kl0tv7lc72hxpsJ3/
        \ny7YYPVGTI7XXDH38lk4AmuQkUvUpiygHOu7qU6YtWOK1tCpnLg2Pk1nwnnRWZeJg\nCLR1kApeUwidbalRUEWrCBUaLoOOthjeZp4HCvughXeDGY4ji2CoXYLN3mYuz0f
        v\nTwIDAQAB\n-----END PUBLIC KEY-----\n"
}
```

❖ We provide the key ID generated above, along with the plaintext and algorithm, then we receive the ciphertext


crypto / encrypt

POST http://localhost:8000/encrypt

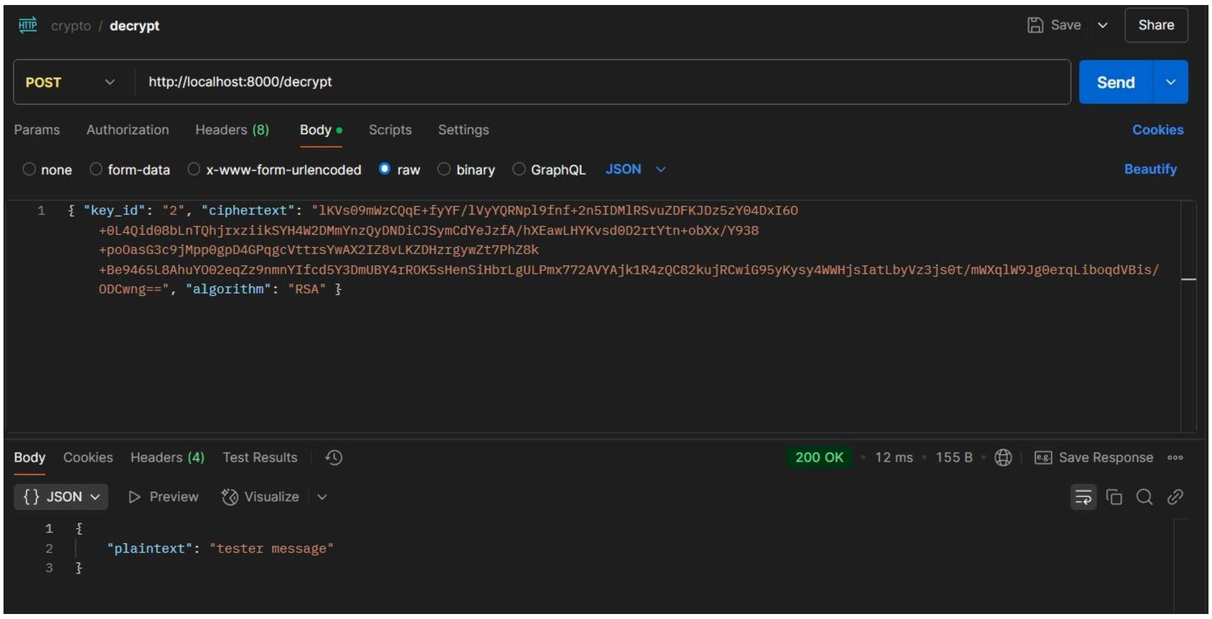
Body: raw, JSON

```
{ "key_id": "2", "plaintext": "tester message", "algorithm": "RSA" }
```

200 OK · 13 ms · 487 B

```
{
    "ciphertext": "lKVs09mWzCQqE+fyYF/lVyYQRNpl9fnf+2n5IDMlRSvuZDFKJDz5zY04DxI6O+0L4Qid08bLnTQhjrxziikSYH4W2DMmYnzQyDNDiCJSymCdYeJzfA/
        hXEawLHYKvsd0D2rtYtn+obXx/Y938+poOasG3c9jMpp0gpD4GPqgcVttrsYwAX2IZ8vLKZDHzrgywZt7PhZ8k
        +Be9465L8AhuYO02eqZz9nmnYIfcd5Y3DmUBY4rROK5sHenSiHbrLgULPmx772AVYAjk1R4zQC82kujRCwiG95yKysy4WWHjsIatLbyVz3js0t/
        mWXqlW9Jg0erqLiboqdVBis/ODCwng=="
}
```

❖ In the decryption stage, we provide the same key ID along with the ciphertext. This will decrypt the message and given the plaintext.


crypto / decrypt

POST http://localhost:8000/decrypt

Body: raw, JSON

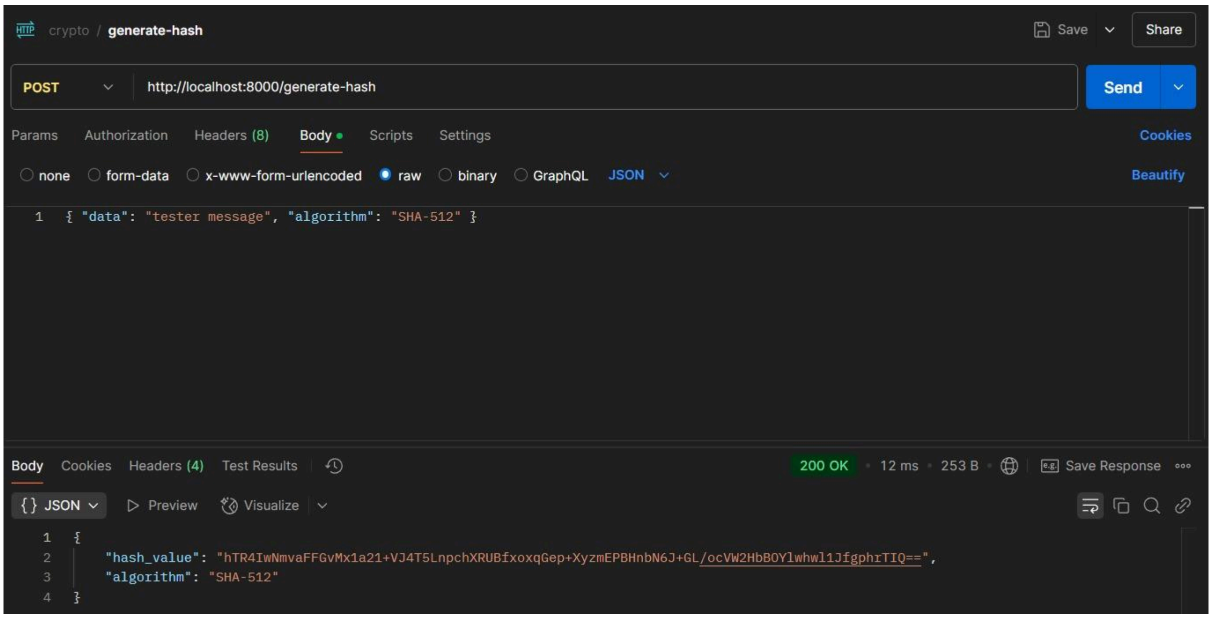
```
{ "key_id": "2", "ciphertext": "lKVs09mWzCQqE+fyYF/lVyYQRNpl9fnf+2n5IDMlRSvuZDFKJDz5zY04DxI6O
    +0L4Qid08bLnTQhjrxziikSYH4W2DMmYnzQyDNDiCJSymCdYeJzfA/hXEawLHYKvsd0D2rtYtn+obXx/Y938
    +poOasG3c9jMpp0gpD4GPqgcVttrsYwAX2IZ8vLKZDHzrgywZt7PhZ8k
    +Be9465L8AhuYO02eqZz9nmnYIfcd5Y3DmUBY4rROK5sHenSiHbrLgULPmx772AVYAjk1R4zQC82kujRCwiG95yKysy4WWHjsIatLbyVz3js0t/mWXqlW9Jg0erqLiboqdVBis/
    ODCwng==", "algorithm": "RSA" }
```

200 OK · 12 ms · 155 B

```
{
    "plaintext": "tester message"
}
```

Hashing demonstration

❖ Here, we use 'tester message' as the data and apply the SHA-512 algorithm. As a result, we receive a hash value.

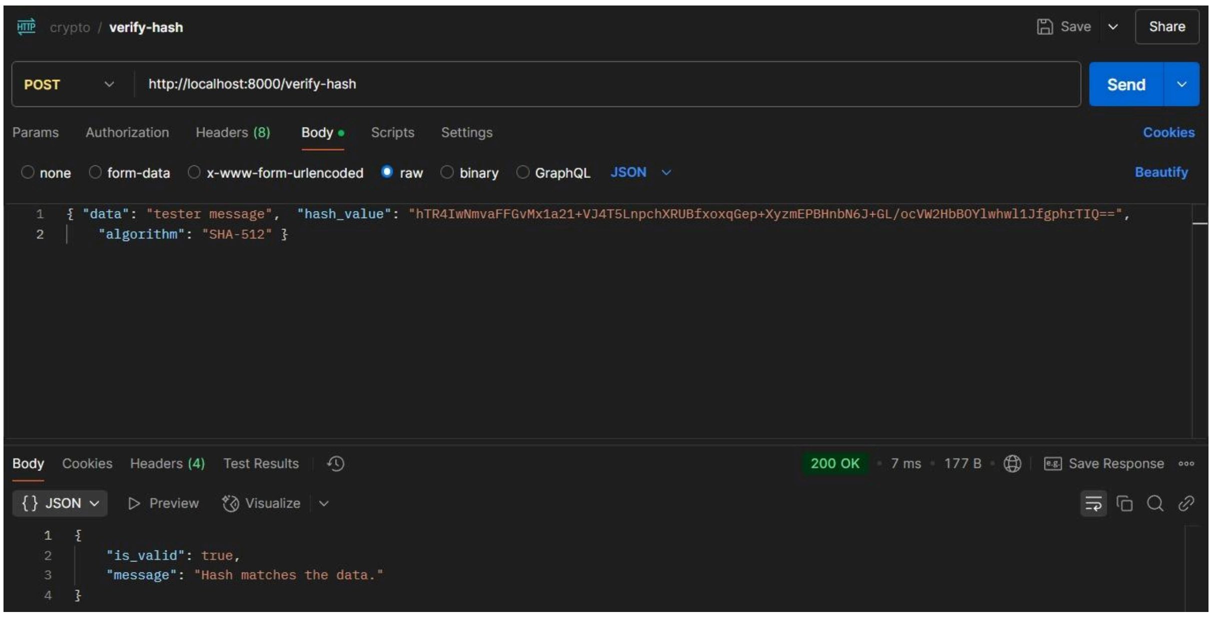

POST http://localhost:8000/generate-hash

```
{ "data": "tester message", "algorithm": "SHA-512" }
```

200 OK · 12 ms · 253 B

```
{
    "hash_value": "hTR4IwNmvaFFGvMx1a21+VJ4T5LnpchXRUBfxoxqGep+XyzmEPBHnbN6J+GL/ocVW2HbBOYlwhwl1JfgphrTIQ==",
    "algorithm": "SHA-512"
}
```

❖ We provide the hash value, algorithm, and data. If the data is the same, the result should be 'True.' As we can see, the output is 'True.'

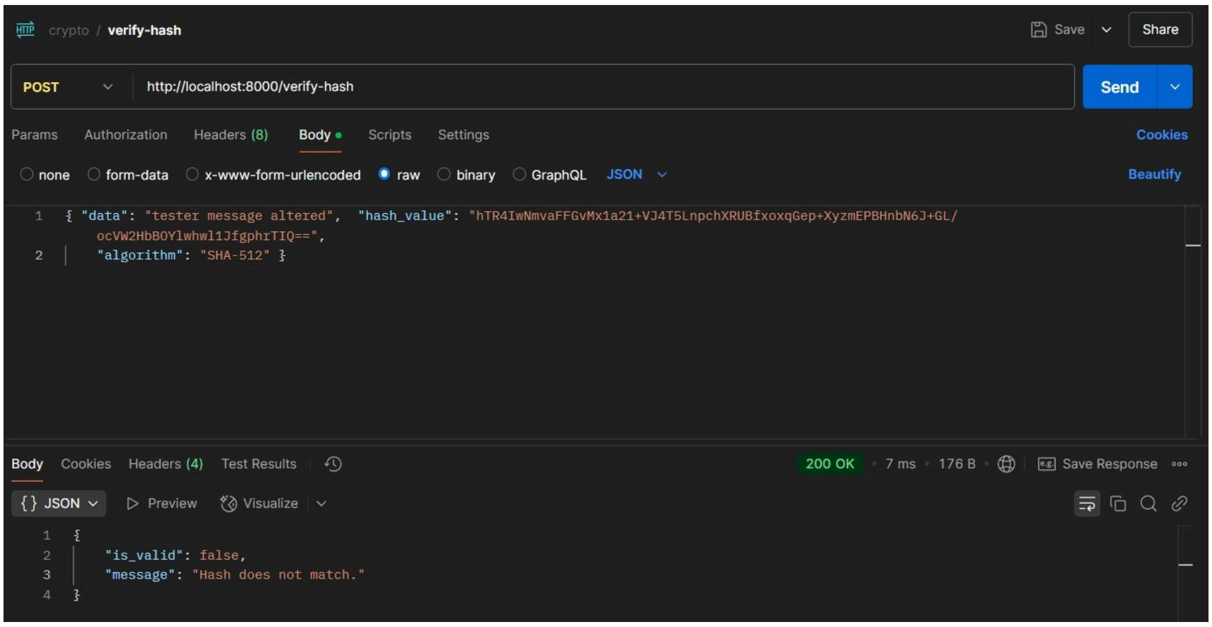

POST http://localhost:8000/verify-hash

```
{ "data": "tester message",  "hash_value": "hTR4IwNmvaFFGvMx1a21+VJ4T5LnpchXRUBfxoxqGep+XyzmEPBHnbN6J+GL/ocVW2HbBOYlwhwl1JfgphrTIQ==",
    "algorithm": "SHA-512" }
```

200 OK · 7 ms · 177 B

```
{
    "is_valid": true,
    "message": "Hash matches the data."
}
```

❖ If the data is not the same, the result should be 'False.' As we can see, the output is 'False.'



POST http://localhost:8000/verify-hash

Body: ○ none ○ form-data ○ x-www-form-urlencoded [✓ raw] ○ binary ○ GraphQL JSON

```
{ "data": "tester message altered",  "hash_value": "hTR4IwNmvaFFGvMx1a21+VJ4T5LnpchXRUBfxoxqGep+XyzmEPBHnbN6J+GL/
    ocVW2HbBOYlwhwl1JfgphrTIQ==",
    "algorithm": "SHA-512" }
```

Body — 200 OK · 7 ms · 176 B — {} JSON

```
{
    "is_valid": false,
    "message": "Hash does not match."
}
```

Github repository links and how to access the APIs (hosted URLs/IPs, parameters, DTO templates

Github repository links : https://github.com/RushenHansana/cyberAPI

URL: http://35.198.225.37:8000/(endpoint)
Ex : http://35.198.225.37:8000/generate-key

Replace (endpoint) with corresponding endpoints.   Eg: generate-key, encrypt, decrypt, generate-hash, verify-hash

1. Key Generation Endpoint

- Endpoint: POST /generate-key
- Description:
  Generates a cryptographic key based on the specified type (AES or RSA) and key size.
- Request Parameters:
  - key_type: A string indicating the type of key (e.g., AES, RSA).
  - key_size: An integer representing the size of the key in bits.
- Response Structure:
  - For AES:

        - key_id: A unique identifier for the generated key.
        - key_value: The cryptographic key encoded in Base64.
    - For RSA:
        - key_id: A unique identifier for the key pair.
        - public_key: The PEM-encoded public key as a string.

2. Encryption Endpoint

- Endpoint: POST /encrypt

- Description:
  Encrypts a plaintext message using the specified cryptographic key and algorithm.

- Request Parameters:

    - key_id: A string that identifies the key to be used for encryption.
    - plaintext: The message to be encrypted.
    - algorithm: A string specifying the encryption algorithm (e.g., AES, RSA).
- Response Structure:

    - ciphertext: The encrypted message encoded in Base64.

3. Decryption Endpoint

- Endpoint: POST /decrypt

- Description:
  Decrypts a Base64-encoded ciphertext back into its original plaintext form using the provided key and algorithm.

- Request Parameters:

    - key_id: A string that identifies the key to be used for decryption.
    - ciphertext: The Base64-encoded encrypted message.
    - algorithm: A string specifying the decryption algorithm (e.g., AES, RSA).
- Response Structure:

    - plaintext: The original, decrypted message as a string.

4. Hash Generation Endpoint

- Endpoint: POST /generate-hash
- Description:
  Computes a cryptographic hash for the provided data using the specified hashing algorithm.
- Request Parameters:
  - data: The input data to be hashed.
  - algorithm: A string specifying the hashing algorithm to use (e.g., SHA-256, SHA-512).
- Response Structure:
  - hash_value: The generated hash, encoded in Base64.
  - algorithm: The hashing algorithm used.

5. Hash Verification Endpoint

- Endpoint: POST /verify-hash
- Description:
  Verifies whether the provided Base64-encoded hash matches the computed hash of the input data using the specified algorithm.
- Request Parameters:
  - data: The original input data to verify.
  - hash_value: The Base64-encoded hash to be verified.
  - algorithm: A string specifying the hashing algorithm used (e.g., SHA-256, SHA-512).
- Response Structure:
  - is_valid: A boolean value indicating whether the provided hash matches the computed hash.
  - message: A string providing additional information about the verification result.

DTO Templates

```
from pydantic import BaseModel

# Key Generation DTOs
class KeyRequest(BaseModel):
        key_type: str
```

```
    key_size: int

class KeyResponseAES(BaseModel):
    key_id: str
    key_value: str

class KeyResponseRSA(BaseModel):
    key_id: str
    public_key: str

# Encryption/Decryption DTOs
class EncryptRequest(BaseModel):
    key_id: str
    plaintext: str
    algorithm: str

class EncryptResponse(BaseModel):
    ciphertext: str

class DecryptRequest(BaseModel):
    key_id: str
    ciphertext: str
    algorithm: str

class DecryptResponse(BaseModel):
    plaintext: str

# Hashing DTOs
class HashRequest(BaseModel):
    data: str
    algorithm: str

class HashResponse(BaseModel):
    hash_value: str
    algorithm: str

# Hash Verification DTOs
class VerifyHashRequest(BaseModel):
    data: str
    hash_value: str
    algorithm: str

class VerifyHashResponse(BaseModel):
    is_valid: bool
    message: str
```