AES Encryption:

Method: **`encrypt_aes`** and **`decrypt_aes`**

- **Description**: AES (Advanced Encryption Standard) is a symmetric encryption algorithm. In this code, AES is used with a 32-byte key (**`aes_key`**), which is randomly generated. The encryption uses Cipher Block Chaining (CBC) mode and PKCS7 padding. CBC mode enhances security by XORing each block of plaintext with the previous ciphertext block before encryption. PKCS7 padding is used to ensure that the plaintext is a multiple of the block size.
- **Usage**: This method is used for encrypting and decrypting data with the same key, making it suitable for scenarios where the same entity is performing both encryption and decryption.

```
# Generate a key for AES
aes_key = os.urandom(32)  # AES key should be either 16, 24, or 32 bytes long

def encrypt_aes(data):
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(data.encode()) + padder.finalize()

    cipher = Cipher(algorithms.AES(aes_key), modes.CBC(os.urandom(16)), backend=default_backend())
    encryptor = cipher.encryptor()
    ct = encryptor.update(padded_data) + encryptor.finalize()

    return ct

def decrypt_aes(encrypted_data):
    cipher = Cipher(algorithms.AES(aes_key), modes.CBC(os.urandom(16)), backend=default_backend())
    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(encrypted_data) + decryptor.finalize()

    unpadder = padding.PKCS7(128).unpadder()
    data = unpadder.update(decrypted_data) + unpadder.finalize()

    return data.decode()
```

RSA Encryption:
Method: **`encrypt_data`** and **`decrypt_data`**

- **Description**: RSA is an asymmetric encryption algorithm. The code generates RSA public and private keys with a key size of 2048 bits. The public key is used for encryption, and the private key is used for decryption. OAEP (Optimal Asymmetric Encryption Padding) with SHA-256 is used for padding, providing additional security against certain attacks.
- **Usage**: This method is ideal for encrypting data that needs to be securely transmitted and decrypted by the intended recipient only.

```
# Generate RSA Key Pair
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)
public_key = private_key.public_key()

# Function to encrypt data
def encrypt_data(public_key, data):
    encrypted = public_key.encrypt(
        data.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return encrypted

# Function to decrypt data
def decrypt_data(private_key, encrypted_data):
    decrypted = private_key.decrypt(
        encrypted_data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return decrypted.decode()
```

Digital Signatures:
Method: **`sign_message`** and **`verify_signature`**

- **Description**: RSA keys are also used for creating and verifying digital signatures. The private key signs a message, and the public key verifies the signature. This uses PSS (Probabilistic Signature Scheme) padding with SHA-256 hashing.
- **Usage**: Digital signatures ensure the authenticity and integrity of messages. They are crucial for scenarios where it's important to verify that a message has not been altered and that it comes from a specific sender.
- **Fernet (Symmetric Key Encryption)**:
- **Method**: Used in **`upload_file`** and **`download_file`** routes.
- **Description**: Fernet is a system for symmetric encryption/decryption using AES in CBC mode with a 128-bit key. It is part of the **`cryptography`** library. The key is generated and used for both encrypting and decrypting data. This method ensures that the data is securely encrypted and can only be decrypted by someone with the same key.
- **Usage**: This is used for encrypting and decrypting file contents in the file upload and download functionalities.

```
def sign_message(private_key, message):
    message = message.encode()
    signature = private_key.sign(
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature

def verify_signature(public_key, message, signature):
    message = message.encode()
    try:
        public_key.verify(
            signature,
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except (ValueError, TypeError):
```

- return False

Fernet (Symmetric Key Encryption):
Method:

- Used in **`upload_file`** and **`download_file`** routes.

Description:

- Fernet is an implementation of symmetric (also known as secret key) encryption and provides strong encryption that is relatively simple to use. It uses AES (Advanced Encryption Standard) in CBC (Cipher Block Chaining) mode with a 128-bit key for encryption, combined with HMAC (Hash-based Message Authentication Code) for ensuring data integrity. The key used for encryption and decryption is the same and is generated by the Fernet class. This key must be kept secret as anyone with access to it can decrypt the data.

Usage:

- In the context of your Flask application, Fernet is used for securely encrypting and decrypting file contents. When a file is uploaded, its contents are encrypted using the Fernet key before being stored. Conversely, when a file is downloaded, its contents are decrypted using the same key. This ensures that the files are stored securely and can only be accessed in their original form by the application or users with the correct key. This method is particularly useful for protecting sensitive data in file storage and transmission scenarios.

```
# Fernet key generation
key = Fernet.generate_key()
cipher_suite = Fernet(key)

# File upload route
@app.route('/upload_file', methods=['GET', 'POST'])
def upload_file():
    # ... [omitted code for brevity]
    if file:
        filename = secure_filename(file.filename)
        file_content = file.read()
        encrypted_content = cipher_suite.encrypt(file_content)
        # ... [omitted code for saving the file]

# File download route
@app.route('/download_file/<filename>', methods=['GET'])
def download_file(filename):
    file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
    # ... [omitted code for brevity]
```

```
    if os.path.exists(file_path):
        with open(file_path, 'rb') as file:
            encrypted_content = file.read()
        decrypted_content = cipher_suite.decrypt(encrypted_content)
        # ... [omitted code for returning the file]
```

Bcrypt (Password Hashing):
Method: Used in the **`register`** and **`login`** routes.

- **Description**: Bcrypt is a password-hashing function. Instead of encrypting passwords, bcrypt hashes them. This is a one-way process, meaning you cannot reverse the hash back to the original password. When a user registers or logs in, the password is hashed (or the hash is compared) using bcrypt.
- **Usage**: This method is used for securely storing user passwords in the database. It ensures that even if the database is compromised, the actual passwords are not easily retrievable.

```
# Registration route

@app.route('/register', methods=['GET', 'POST'])

def register():

    if request.method == 'POST':

        username = request.form['username']

        password = request.form['password'].encode('utf-8')

        password_hash = bcrypt.hashpw(password, bcrypt.gensalt())

        # ... [omitted code for saving the user]


# Login route

@app.route('/login', methods=['GET', 'POST'])

def login():

    if request.method == 'POST':

        username = request.form['username']

        password = request.form['password'].encode('utf-8')

        # ... [omitted code for retrieving the user]

        if user and bcrypt.checkpw(password, user['password_hash']):
```

```
# ... [omitted code for successful login]
```