Implementing Endpoint for Flows

This guide explains how to implement an Endpoint to power WhatsApp Flows.

> Please note that Flows with an endpoint are subject to both reliability and performance requirements. For additional information, refer to the Flows Health and Monitoring guide.

Endpoint Examples

Basic Endpoint Example

We have created an endpoint example in Node.js that you can clone to create your own endpoint and quickly prototype your flow. Follow the instructions in the README.md file to get started. You can clone the example code from Github and run it in any environment you prefer.

"Book an Appointment" Endpoint Example

We also provide endpoint code that can be used along with the "Book an Appointment" flow JSON template to complete the flow from start to finish. Follow the instructions in the README.md file to get started. You can clone the example code from Github and run it in any environment you prefer.

Set up an Endpoint

Endpoints provide dynamic data for the screens and control routing, i.e. upon screen submission, the flow can make a request to the endpoint to get the name of the next screen and the data to display on it. Also, the endpoint can instruct the flow to terminate and control whether an outgoing

Setting up an endpoint consists of the following steps:

1. Create a key pair and upload and sign the public key using On-Prem or the Cloud API.
2. Setup the HTTP endpoint.
3. Implement Payload Encryption/Decryption.
4. Link the endpoint to your flow:
    - specify `data_api_version` in the Flow JSON and configure endpoint url, see reference for more details.
    - if the Flow was created through WhatsApp Manager, connect Meta App to it in "Edit Flow" page, see Flow Builder UI for more details.

After setting up an endpoint, implement it's logic to handle requests:

- Data Exchange
- Error Notification
- Health Check

Upload Public Key

Data exchanged with an endpoint is encrypted using a combination of symmetric and asymmetric encryption. You should have a key pair to enable encryption and upload the public key. It will be automatically signed along the way.

> Since Solution Partners manage multiple businesses, it's recommended to have a dedicated endpoint and an encryption key pair for each WABA.

Please ensure you have the versions specified below when signing the business public key.

Tool	Required Version
On-Prem API	v2.51.x

Uploading and signing the business public key is different for On-Prem and Cloud, so depending on your API client please refer to the appropriate guide:

Sign and upload the business public key

You will need to re-upload public key in the following cases:

- When you re-register your number on On-Prem or Cloud API.
- When you migrate your number from On-Prem to Could API or vice versa.
- When you start receiving webhooks with alerts about client side errors `public-key-missing` or `public-key-signiture-verification`.

Setup HTTP Endpoint

When the flow will need to exchange the data with your endpoint, it will make an HTTP request to it. You should set up a server and provide it's url while configuring the flow, for example:

`https://business.com/scheduleappointment`

Your server must be enabled to receive and process `POST` requests, use HTTPS and have a valid TLS/SSL certificate installed. This certificate does not have to be used in payload encryption/decryption.

Implement Encryption/Decryption

The body of each request contains the encrypted payload and has the following form:

Sample Endpoint Request

```
{
    encrypted_flow_data: "<ENCRYPTED FLOW DATA>",
    encrypted_aes_key: "<ENCRYPTED_AES_KEY>",
    initial_vector: "<INITIAL VECTOR>"
 }
```

Parameter	Description
`encrypted_flow_data` *string*	**Required.** The encrypted request payload.

`encrypted_aes_key` *string*	**Required.** The encrypted 128-bit AES key.
`initial_vector` *string*	**Required.** The 128-bit initialization vector.

After processing the decrypted request, create a response and encrypt it before sending it back to the WhatsApp client. Encrypt the payload using the AES key received in the request and send it back as a Base64 string.

You can reference the below examples of how to decrypt and encrypt.

> If request can not be decrypted, endpoint should return HTTP 421 response status code (see Business Endpoint Error Codes for more details).

Sample Endpoint Response

```
curl -i -H "Content-Type: application/json" -X POST -d '{
"encrypted_flow_data":"4Wor0bpfvrNqnkH+XQZLn3HnU2Zi7hG\\/UHjISS93Fzn9J7youssaL
"encrypted_aes_key":"<ufA0fXD1WzMS4f2aCyr2JI4KtV2X+puen78fLjjt7mI+NqITDCypLOlc
"initial_vector":"<G\\/1rq1naEOMR4TJHFvIs\\/Q==.>"
}' 'https://business.com/testing_flow'

HTTP/2 200
content-type: text/plain
content-length: 232
date: Wed, 06 Jul 2022 14:03:03 GMT

yZcJQaH3AqfzKgjn64vAcASaJrOMN27S6CESyU68WN/cDCP6abskoMa/pPjszXGKyyh/23lw84HW6Z
```

Implement Endpoint Logic

> Endpoint-powered Flows should avoid using the endpoint when it is not needed. This will reduce the latency for consumers and simplify the development of the Flow.

1. User opens the flow:
    - If `flow_action` field in the parameters that you pass to On-Prem or CAPI when sending a flow message is `data_exchange`;
    - See Data Exchange Request for details.
    - Tip: If the first screen of your flow does not have any parameters or the parameters are known when the message is sent, consider omitting `flow_action` to avoid calling your endpoint. You can supply parameters by using the `flow_action_payload.data` message field instead.

2. User submits the screen:
    - If `name` attribute specified inside `on-click-action` field in Flow JSON is `data_exchange`;
    - See Data Exchange Request for details.
    - Tip: If the next screen and its data are known already, consider setting the `on-click-action` name to `navigate` to avoid calling your endpoint.

3. User presses back button on the screen:
    - If `refresh_on_back` attribute specified in Flow JSON for the screen is `true`;
    - See Data Exchange Request for details.
    - Tip: If custom behavior when pressing the back button is not needed, consider omitting `refresh_on_back` to avoid calling your endpoint.

4. User changes the value of a component:
    - If `on-select-action` for the component is defined in Flow JSON.

5. Your endpoint replied with invalid content to the previous request (e.g. required field was missing), in this case consumer client will send asynchronous error notification request:
    - See Error Notification Request.

6. Periodical health check from WhatsApp:
    - See Health Check Request

Data Exchange Request

Data exchange request is used to query the name of the next screen and data required to render it. Decrypted payload of the data exchange request will have below format.

Sample Data Exchange Request Payload

```
    "action": "<ACTION_NAME>",
    "screen": "<SCREEN_NAME>",
    "data": {
      "prop_1": "value_1",
       …
      "prop_n": "value_n"
    },
   "flow_token": "<FLOW-TOKEN>"
}
```

Parameter	Description
`version` *string*	**Required.** Value must be set to `3.0`.
`screen` *string*	**Required.** If `action` is set to `INIT` or `BACK`, this field may not be populated. (Note: "SUCCESS" is a reserved name and cannot be used by any screens.)
`action` *string*	**Required.** Defines the type of the request. For data exchange request there are multiple choices depending on when the trigger: • `INIT` if the request is triggered when opening the Flow • `BACK` if the request is triggered when pressing "back" • `data_exchange` if the request is triggered when submitting the screen
`data` *object*	**Required.** An object passing arbitrary key-value data as a JSON object. If `action` is set to `INIT` or `BACK`, this field may not be populated. `<key>` *string, boolean, number, object, array* - `<value>`
`flow_token` *string*	**Required.** A Flow token generated and sent by you as part of the Flow message. The flow token is similar to a session identifier commonly used in web applications. It should be generated using established best practices (e.g.

	an endpoint.
`flow_token_signature` *string*	Please note that flow_token_signature will only be sent with flows version >= 7.3 and data_api_version >=4.0. A Flow token signature is generated and sent by flows as part of the data exchange request payload. The flow_token_signature is a JSON Web Token (JWT) created by flows to securely sign the flow token using the Meta app secret as the secret key. You can choose to use this signature to verify the authenticity of the flow token. (see Flow token Signature for more details)

After the request is received and decrypted, your business logic processes the request and determines which screen and data is to be sent back to the WhatsApp client. If user needs to be redirected to the next screen, next screen response payload should be sent. If flow needs to be terminated (e.g. because it's completed), final response payload should be sent.

Next Screen Response Payload for Data Exchange Request

The following response payload is what the data channel needs to send back to the Whatsapp client during each data exchange, except the last one:

```
{
    "screen": "<SCREEN_NAME>",
    "data": {
      "property_1": "value_1",
       ...
      "property_n": "value_n",
      "error_message": "<ERROR-MESSAGE>"
    }
}
```

This will redirect the user to `<SCREEN_NAME>` and will trigger a snackbar error with the `error_message` present.

Parameter	Description
`screen` *string*	**Required.** The screen to be rendered once the data exchange is complete.
`data` *object*	**Required.** A JSON of properties and its values to render the screen after data exchange is complete. • `error_message` *string* – If a bad request was sent from the WhatsApp client to you, the error message will be defined here. • `<key>` *string, boolean, number, object, array* – `<value>`: A property and its respective value which can be referenced in screen layout in Flow JSON.

Final Response Payload

To trigger flow completion, send the following response to the data exchange request:

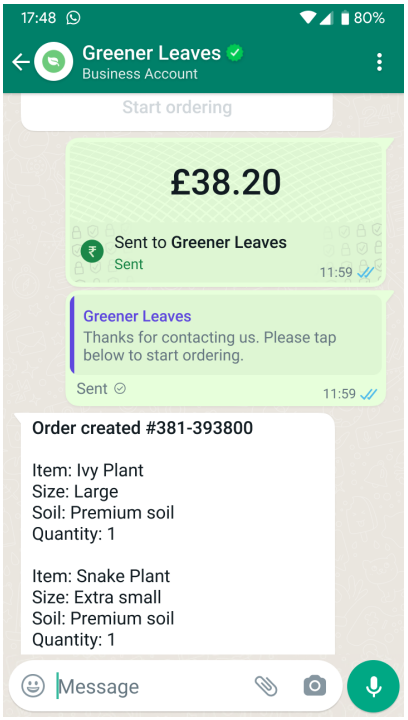
```
{
    "screen": "SUCCESS",
    "data": {
        "extension_message_response": {
            "params": {
                "flow_token": "<FLOW_TOKEN>",
                "optional_param1": "<value1>",
                "optional_param2": "<value2>"
            }
        }
    }
}
```

Parameter	Description

`screen` *string*	Value must be `SUCCESS`
`data.extension_message_response.params` *object*	A JSON with data which will be included to the flow completion message (see Response Message Webhook for more details)
`data.extension_message_response.params.flow_token` *string*	**Required.** Flow token generated by a business signifying a session or a user flow

As a result, flow will be closed and a flow response message will be sent to the chat. See Response Message Webhook for additional details.

It is highly recommended that you manually send a summary message to the chat with consumer in response, such as the one below:



If you need parameters from the data channel for the message content, you can send them in the `params` field in addition to `flow_token`. All these parameters are forwarded to the messages

Error Notification Request

If you send a bad response payload to the WhatsApp client, you will receive a payload detailing the error and the error type. This provides you more visibility on failed client interactions so you can respond appropriately.

Sample Error Notification Request Payload

```
{
    "version": "<VERSION>",
    "flow_token": "<FLOW-TOKEN>",
    "action": "data_exchange | INIT",
    "data": {
        "error": "<ERROR-KEY>",
        "error_message": "<ERROR-MESSAGE>"
    }
}
```

Parameter	Description
`version` *string*	**Required.** 3.0
`screen` *string*	**Required.** Screen name where bad intermediate response payload was sent
`flow_token` *string*	**Required.** A flow token generated by your business
`action` *string*	**Required.** Will be `"data_exchange"` or `"INIT"`

`data` *object*	**Required.** `data` object representing the error. • `data.error_key` *string* – The error code for the invalid payload. • `data.error_message` *string* – The error message associated with the error code.

Error Notification Response Payload

Send the following response payload to indicate that error notification was acknowledged:

```
{
    "data": {
        "acknowledged": true
    }
}
```

Health Check Request

Endpoints should be able to respond to health check requests. WhatsApp may periodically send health check requests to the endpoints used by published flows.

Sample Health Check Request Payload

```
{
    "version": "3.0",
    "action": "ping"
}
```

You should generate the following response payload:

Health Check Response Payload

```
        "status": "active"
    }
}
```

Request Signature Validation

> When creating your app, decide who owns it and the endpoint, whether it is you or the Solution Partner. In order to prevent sharing of the app secret, the owners must be the same for both to use the app secret to validate the payload.

You can verify that request is coming from Meta by checking signature which is generated using an app secret from the app connected to the flow. It is inferred from the user token when the flow is created through API, or selected manually in the Flow Builder when endpoint is added to the flow.

Signature and Header Format

We sign all endpoint requests with a **SHA256** signature and include the signature in the request's `X-Hub-Signature-256` header, preceded with `sha256=`.

To validate the signature:

1. Generate a **SHA256** signature using the payload and your app secret.
2. Compare your signature to the signature in the `X-Hub-Signature-256` header (everything after `sha256=`). If the signatures match, the payload is genuine.

If validation failes, appropriate HTTP code should be returned. Please see the Business Endpoint Error Codes for more details.

Resetting the App Secret

If you need to reset the app secret without any downtime and without turning off payload validation, you may use below approach:

1. Allow the old app secret to continue generating the **SHA256** signature for *X* hours
2. Temporary consider `X-Hub-Signature-256` header to be correct if it can be validated using either old or new app secret.
3. After *X* hours (not earlier), consider the signature in the `X-Hub-Signature-256` header

Flow token signature

flow_token_signature for enhanced endpoint authentication

To enhance the security and integrity of flow interactions, we are introducing a parameter called flow_token_signature. This signature allows businesses to verify the authenticity of the flow token received in the message payload.

What is flow_token_signature?

The flow_token_signature is a JSON Web Token (JWT) created bty flows to securely sign the flow token using the Meta app secret as the secret key. This JWT includes a header specifying the signing algorithm (HS256), a payload containing the flow token. It enables businesses to verify the authenticity and integrity of the flow token received in the message payload, ensuring the token has not been tampered with during transmission.

How to use?

The flow_token_signature is a parameter sent by flows and included in the data exchange request payload. Businesses can choose to use this signature to verify the authenticity of the flow token. To do so, businesses need to decode and verify the JWT using the Meta app secret, which is known to them. Once decoded, businesses can confirm that the flow_token value inside the token matches the expected value, ensuring the token has not been tampered with during transmission.

Request Decryption and Encryption

The incoming request body is encrypted, you need to decrypt it first, then you need to encrypt the server response before returning it to the client.

> You can find code examples of decryption/encryption in various programming languages in the Code Examples section.

For data_api_version "3.0" you should follow below instructions to decrypt request payload:

- decrypt resulting byte array with the private key corresponding to the uploaded public key using RSA/ECB/OAEPWithSHA-256AndMGF1Padding algorithm with SHA256 as a hash function for MGF1;
- as a result, you'll get a 128-bit payload encryption key.

2. decrypt request payload from `encrypted_flow_data` field:
    - decode base64-encoded field content to get encrypted byte array;
    - decrypt encrypted byte array using AES-GCM algorithm, payload encryption key and initialization vector passed in `initial_vector` field (which is base64-encoded as well and should be decoded first). Note that the 128-bit authentication tag for the AES-GCM algorithm is appended to the end of the encrypted array.
    - result of above step is UTF-8 encoded clear request payload.

For data_api_version "3.0" you should follow below instructions to encrypt the response:

- encode response payload string to response byte array using UTF-8
- prepare initialization vector for response encryption by inverting all bits of the initialization vector used for request payload encryption
- encrypt response byte array using AES-GCM algorithm with the following parameters:
    - secret key - payload encryption key from request decryption stage
    - initialization vector for response encryption from above step
    - empty AAD (additional authentication data) - many libraries assume this by default, check the documentation of the library in use
    - 128-bit (16 byte) length for authentication tag - many libraries assume this by default, check the documentation of the library in use
- append authentication tag generated during encryption to the end of the encryption result
- encode the whole output as base64 string and send it in the HTTP response body as plain text

Handling Decryption Errors

If you can't decrypt a request, you should send appropriate HTTP response code to force mobile client to re-download public key and retry the query. See endpoint error codes for additional details.

Code Examples

The below code examples are only meant to demonstrate the encryption/decryption implementation and are not production ready.

Python Django Example

Here's a full code sample to handle a request with decryption/encryption in Python with Django framework. Note that the response is sent as a plain text string.

```python
import json
import os
from base64 import b64decode, b64encode
from cryptography.hazmat.primitives.asymmetric.padding import OAEP, MGF1, hash
from cryptography.hazmat.primitives.ciphers import algorithms, Cipher, modes
from cryptography.hazmat.primitives.serialization import load_pem_private_key
from django.http import HttpResponse
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt

# Load the private key string
PRIVATE_KEY = os.environ.get('PRIVATE_KEY')
# Example:
# '''-----BEGIN RSA PRIVATE KEY-----
# MIIE...
# ...
# ...AQAB
# -----END RSA PRIVATE KEY-----'''


@csrf_exempt
def data(request):
    try:
        # Parse the request body
        body = json.loads(request.body)

        # Read the request fields
        encrypted_flow_data_b64 = body['encrypted_flow_data']
        encrypted_aes_key_b64 = body['encrypted_aes_key']
        initial_vector_b64 = body['initial_vector']

        decrypted_data, aes_key, iv = decrypt_request(
            encrypted_flow_data_b64, encrypted_aes_key_b64, initial_vector_b64
        print(decrypted_data)
```

```
            "data": {
                "some_key": "some_value"
            }
        }

        # Return the response as plaintext
        return HttpResponse(encrypt_response(response, aes_key, iv), content_t
    except Exception as e:
        print(e)
        return JsonResponse({}, status=500)


def decrypt_request(encrypted_flow_data_b64, encrypted_aes_key_b64, initial_ve
    flow_data = b64decode(encrypted_flow_data_b64)
    iv = b64decode(initial_vector_b64)

    # Decrypt the AES encryption key
    encrypted_aes_key = b64decode(encrypted_aes_key_b64)
    private_key = load_pem_private_key(
        PRIVATE_KEY.encode('utf-8'), password=None)
    aes_key = private_key.decrypt(encrypted_aes_key, OAEP(
        mgf=MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=

    # Decrypt the Flow data
    encrypted_flow_data_body = flow_data[:-16]
    encrypted_flow_data_tag = flow_data[-16:]
    decryptor = Cipher(algorithms.AES(aes_key),
                       modes.GCM(iv, encrypted_flow_data_tag)).decryptor()
    decrypted_data_bytes = decryptor.update(
        encrypted_flow_data_body) + decryptor.finalize()
    decrypted_data = json.loads(decrypted_data_bytes.decode("utf-8"))
    return decrypted_data, aes_key, iv


def encrypt_response(response, aes_key, iv):
    # Flip the initialization vector
    flipped_iv = bytearray()
    for byte in iv:
        flipped_iv.append(byte ^ 0xFF)

    # Encrypt the response data
    encryptor = Cipher(algorithms.AES(aes_key),
                       modes.GCM(flipped_iv)).encryptor()
    return b64encode(
        encryptor.update(json.dumps(response).encode("utf-8")) +
        encryptor.finalize() +
        encryptor.tag
    ).decode("utf-8")
```

Here's a full code sample to handle a request with decryption/encryption in NodeJS with Express framework. Note that the response is sent as a plain text string.

```
import express from "express";
import crypto from "crypto";

const PORT = 3000;
const app = express();
app.use(express.json());

const PRIVATE_KEY = process.env.PRIVATE_KEY as string;
/*
Example:
```-----BEGIN RSA PRIVATE KEY-----
MIIE...
...
...AQAB
-----END RSA PRIVATE KEY-----```
*/

app.post("/data", async ({ body }, res) => {
  const { decryptedBody, aesKeyBuffer, initialVectorBuffer } = decryptRequest(
    body,
    PRIVATE_KEY,
  );

  const { screen, data, version, action } = decryptedBody;
  // Return the next screen & data to the client
  const screenData = {
    screen: "SCREEN_NAME",
    data: {
      some_key: "some_value",
    },
  };

  // Return the response as plaintext
  res.send(encryptResponse(screenData, aesKeyBuffer, initialVectorBuffer));
});

const decryptRequest = (body: any, privatePem: string) => {
  const { encrypted_aes_key, encrypted_flow_data, initial_vector } = body;

  // Decrypt the AES key created by the client
  const decryptedAesKey = crypto.privateDecrypt(
    {
      key: crypto.createPrivateKey(privatePem),
      padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
```
```

```
  );

  // Decrypt the Flow data
  const flowDataBuffer = Buffer.from(encrypted_flow_data, "base64");
  const initialVectorBuffer = Buffer.from(initial_vector, "base64");

  const TAG_LENGTH = 16;
  const encrypted_flow_data_body = flowDataBuffer.subarray(0, -TAG_LENGTH);
  const encrypted_flow_data_tag = flowDataBuffer.subarray(-TAG_LENGTH);

  const decipher = crypto.createDecipheriv(
    "aes-128-gcm",
    decryptedAesKey,
    initialVectorBuffer,
  );
  decipher.setAuthTag(encrypted_flow_data_tag);

  const decryptedJSONString = Buffer.concat([
    decipher.update(encrypted_flow_data_body),
    decipher.final(),
  ]).toString("utf-8");

  return {
    decryptedBody: JSON.parse(decryptedJSONString),
    aesKeyBuffer: decryptedAesKey,
    initialVectorBuffer,
  };
};

const encryptResponse = (
  response: any,
  aesKeyBuffer: Buffer,
  initialVectorBuffer: Buffer,
) => {
  // Flip the initialization vector
  const flipped_iv = [];
  for (const pair of initialVectorBuffer.entries()) {
    flipped_iv.push(~pair[1]);
  }
  // Encrypt the response data
  const cipher = crypto.createCipheriv(
    "aes-128-gcm",
    aesKeyBuffer,
    Buffer.from(flipped_iv),
  );
  return Buffer.concat([
    cipher.update(JSON.stringify(response), "utf-8"),
    cipher.final(),
```

```
app.listen(PORT, () => {
  console.log(`App is listening on port ${PORT}!`);
});
```

PHP Slim Example

Here's a full code sample to handle a request with decryption/encryption in PHP with Slim framework. Note that the response is sent as a plain text string.

```
<?php
use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Message\ServerRequestInterface as Request;
use phpseclib3\Crypt\RSA;
use phpseclib3\Crypt\AES;

require __DIR__ . '/vendor/autoload.php';

$app = Slim\Factory\AppFactory::create();
$app->post('/data', function (Request $request, Response $response) {
    $body = json_decode($request->getBody()->getContents(), true);

    $privatePem = getenv('PRIVATE_KEY');
    /*
    Example:
    ```-----BEGIN RSA PRIVATE KEY-----
    MIIE...
    ...
    ...AQAB
    -----END RSA PRIVATE KEY-----```
    */
    $decryptedData = decryptRequest($body, $privatePem);

    // Return the next screen & data to client
    $screen = [
        "screen" => "SCREEN_NAME",
        "data" => [
            "some_key" => "some_value"
        ]
    ];
    $resBody = encryptResponse($screen, $decryptedData['aesKeyBuffer'], $decry
    // Return the response as plaintext
    $response->getBody()->write($resBody);
    return $response;
});
```
```

```php
    $encryptedAesKey = base64_decode($body['encrypted_aes_key']);
    $encryptedFlowData = base64_decode($body['encrypted_flow_data']);
    $initialVector = base64_decode($body['initial_vector']);

    // Decrypt the AES key created by the client
    $rsa = RSA::load($privatePem)
        ->withPadding(RSA::ENCRYPTION_OAEP)
        ->withHash('sha256')
        ->withMGFHash('sha256');

    $decryptedAesKey = $rsa->decrypt($encryptedAesKey);
    if (!$decryptedAesKey) {
        throw new Exception('Decryption of AES key failed.');
    }

    // Decrypt the Flow data
    $aes = new AES('gcm');
    $aes->setKey($decryptedAesKey);
    $aes->setNonce($initialVector);
    $tagLength = 16;
    $encryptedFlowDataBody = substr($encryptedFlowData, 0, -$tagLength);
    $encryptedFlowDataTag = substr($encryptedFlowData, -$tagLength);
    $aes->setTag($encryptedFlowDataTag);

    $decrypted = $aes->decrypt($encryptedFlowDataBody);
    if (!$decrypted) {
        throw new Exception('Decryption of flow data failed.');
    }

    return [
        'decryptedBody' => json_decode($decrypted, true),
        'aesKeyBuffer' => $decryptedAesKey,
        'initialVectorBuffer' => $initialVector,
    ];
}

function encryptResponse($response, $aesKeyBuffer, $initialVectorBuffer)
{
    // Flip the initialization vector
    $flipped_iv = ~$initialVectorBuffer;

    // Encrypt the response data
    $cipher = openssl_encrypt(json_encode($response), 'aes-128-gcm', $aesKeyBu
    return base64_encode($cipher . $tag);
}

$app->run();
```

Here's a full code sample to handle a request with decryption/encryption in Java 8+ using simple-json library:

> Please note that this example requires private key to be in unencrypted PKCS8 format, which is normally indicated by `-----BEGIN PRIVATE KEY-----` at the beginning of the file.
>
> Depending on the way and exact software which you used to generate private key, you may need to convert it to the required format first.
>
> For example, if your key is in PKCS#1 format (starts with `-----BEGIN RSA PRIVATE KEY-----`) or PKCS#8 encrypted format (starts with `-----BEGIN ENCRYPTED PRIVATE KEY-----`), you can use below command to convert it to the unencrypted PKCS#8:
>
> openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in private.pem -out private_unencrypted_pkcs8.pem

```
package org.example;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;

import javax.crypto.Cipher;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.OAEPParameterSpec;
import javax.crypto.spec.PSource;
import javax.crypto.spec.SecretKeySpec;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.security.GeneralSecurityException;
import java.security.KeyFactory;
import java.security.interfaces.RSAPrivateKey;
import java.security.spec.MGF1ParameterSpec;
import java.security.spec.PKCS8EncodedKeySpec;
import java.util.Base64;

public class App {
```

```
        public DecryptionInfo(String clearPayload, byte[] clearAesKey) {
            this.clearPayload = clearPayload;
            this.clearAesKey = clearAesKey;
        }
    }

    private static final int AES_KEY_SIZE = 128;
    private static final String KEY_GENERATOR_ALGORITHM = "AES";
    private static final String AES_CIPHER_ALGORITHM = "AES/GCM/NoPadding";
    private static final String RSA_ENCRYPT_ALGORITHM = "RSA/ECB/OAEPWithSHA-2
    private static final String RSA_MD_NAME = "SHA-256";
    private static final String RSA_MGF = "MGF1";

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress(3000), 0);
        server.createContext("/data", new EndpointHandler());
        server.setExecutor(null);
        server.start();
        System.out.print("Server started on " + server.getAddress());
    }

    static class EndpointHandler implements HttpHandler {
        @Override
        public void handle(HttpExchange t) throws IOException {
            String response;
            int responseCode;
            try {
                final JSONParser parser = new JSONParser();
                final JSONObject requestJson = (JSONObject) parser.parse(new I
                final byte[] encrypted_flow_data = Base64.getDecoder().decode(
                final byte[] encrypted_aes_key = Base64.getDecoder().decode((S
                final byte[] initial_vector = Base64.getDecoder().decode((Stri
                final DecryptionInfo decryptionInfo = decryptRequestPayload(en
                final JSONObject clearRequestData = (JSONObject) parser.parse(
                final String clearResponse = String.format("{\"screen\":\"SCRE
                response = encryptAndEncodeResponse(clearResponse, decryptionI
                responseCode = 200;
            } catch (Exception ex) {
                response = "Processing error: " + ex.getMessage();
                responseCode = 500;
            }
            t.getResponseHeaders().add("Content-Type", "text/plain; charset=UT
            final byte[] responseBytes = response.getBytes();
            t.sendResponseHeaders(responseCode, responseBytes.length);
            OutputStream os = t.getResponseBody();
            os.write(response.getBytes());
            os.close();
```

```java
private static DecryptionInfo decryptRequestPayload(byte[] encrypted_flow_
    final RSAPrivateKey privateKey = readPrivateKeyFromPkcs8UnencryptedPem
    final byte[] aes_key = decryptUsingRSA(privateKey, encrypted_aes_key);
    return new DecryptionInfo(decryptUsingAES(encrypted_flow_data, aes_key
}

private static String decryptUsingAES(final byte[] encrypted_payload, fina
    final GCMParameterSpec paramSpec = new GCMParameterSpec(AES_KEY_SIZE,
    final Cipher cipher = Cipher.getInstance(AES_CIPHER_ALGORITHM);
    cipher.init(Cipher.DECRYPT_MODE, new SecretKeySpec(aes_key, KEY_GENERA
    final byte[] data = cipher.doFinal(encrypted_payload);
    return new String(data, StandardCharsets.UTF_8);
}

private static byte[] decryptUsingRSA(final RSAPrivateKey privateKey, fina
    final Cipher cipher = Cipher.getInstance(RSA_ENCRYPT_ALGORITHM);
    cipher.init(Cipher.DECRYPT_MODE, privateKey, new OAEPParameterSpec(RSA
    return cipher.doFinal(payload);
}

private static RSAPrivateKey readPrivateKeyFromPkcs8UnencryptedPem(String
    final String prefix = "-----BEGIN PRIVATE KEY-----";
    final String suffix = "-----END PRIVATE KEY-----";
    String key = new String(Files.readAllBytes(new File(filePath).toPath()
    if (!key.contains(prefix)) {
        throw new IllegalStateException("Expecting unencrypted private key
    }
    String privateKeyPEM = key.replace(prefix, "").replaceAll("[\\r\\n]",
    byte[] encoded = Base64.getDecoder().decode(privateKeyPEM);
    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec(encoded);
    return (RSAPrivateKey) keyFactory.generatePrivate(keySpec);
}

private static String encryptAndEncodeResponse(final String clearResponse,
    final GCMParameterSpec paramSpec = new GCMParameterSpec(AES_KEY_SIZE,
    final Cipher cipher = Cipher.getInstance(AES_CIPHER_ALGORITHM);
    cipher.init(Cipher.ENCRYPT_MODE, new SecretKeySpec(aes_key, KEY_GENERA
    final byte[] encryptedData = cipher.doFinal(clearResponse.getBytes(Sta
    return Base64.getEncoder().encodeToString(encryptedData);
}

private static byte[] flipIv(final byte[] iv) {
    final byte[] result = new byte[iv.length];
    for (int i = 0; i < iv.length; i++) {
        result[i] = (byte) (iv[i] ^ 0xFF);
    }
    return result;
```

C# Example

Here's a full code sample to handle a request with decryption/encryption in C#. Note that the response is sent as a plain text string. View the full project code on github.

```
using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.OpenSsl;
using Org.BouncyCastle.Security;

var app = WebApplication.CreateBuilder(args).Build();
var PRIVATE_KEY = Environment.GetEnvironmentVariable("PRIVATE_KEY") ?? throw n
var PASSPHRASE = Environment.GetEnvironmentVariable("PASSPHRASE") ?? throw new

app.MapPost("/", (EndpointPayload body) =>
{
    var decrypted = EncryptionUtils.DecryptRequest(body.encrypted_aes_key, bod

    // Example to read decrypted fields
    var action = decrypted.decryptedBody.GetProperty("action").GetString();

    // Return the next screen & data to client
    var response = new { screen = "SCREEN_NAME", data = new { some_key = "some
    var encryptedResponse = EncryptionUtils.EncryptResponse(response, decrypte

    // Return the response as plaintext
    return Results.Content(encryptedResponse, "text/plain");

})
.WithName("PostEndpointData");

app.Run();

record EndpointPayload(string encrypted_aes_key, string encrypted_flow_data, s

public class EncryptionUtils
{
    const int TAG_LENGTH = 16;

    public static (dynamic decryptedBody, byte[] aesKeyBytes, byte[] initialVe
```

```
        {
            // Load the private key from PEM
            var pemReader = new PemReader(new StringReader(privatePem), new Pa
            if (pemReader.ReadObject() is AsymmetricCipherKeyPair keyPair)
            {
                // Extract the private key parameters
                var privateKey = keyPair.Private as RsaPrivateCrtKeyParameters
                if (privateKey == null)
                {
                    throw new CryptographicException("The provided PEM does no
                }

                // Convert Bouncy Castle RSA key parameters to .NET-compatible
                var rsaParams = DotNetUtilities.ToRSAParameters(privateKey);
                // Import into .NET RSA
                rsa.ImportParameters(rsaParams);
            }
            else
            {
                throw new CryptographicException("The provided PEM is not a va
            }

            // Decrypt the AES key created by the client
            byte[] encryptedAesKeyBytes = Convert.FromBase64String(encryptedAe
            byte[] aesKeyBytes = rsa.Decrypt(encryptedAesKeyBytes, RSAEncrypti

            // Decrypt the Flow data
            byte[] initialVectorBytes = Convert.FromBase64String(initialVector
            byte[] flowDataBytes = Convert.FromBase64String(encryptedFlowData)
            byte[] plainTextBytes = new byte[flowDataBytes.Length - TAG_LENGTH

            var cipher = new GcmBlockCipher(new AesEngine());
            var parameters = new AeadParameters(new KeyParameter(aesKeyBytes),
            cipher.Init(false, parameters);
            var offset = cipher.ProcessBytes(flowDataBytes, 0, flowDataBytes.L
            cipher.DoFinal(plainTextBytes, offset);

            string decryptedJsonString = Encoding.UTF8.GetString(plainTextByte
            dynamic decryptedBody = JsonSerializer.Deserialize<dynamic>(decryp
            return (decryptedBody: decryptedBody, aesKeyBytes: aesKeyBytes, in
        }
    }

    public static string EncryptResponse(dynamic response, byte[] aesKeyBytes,
    {
        // Flip the initialization vector
        byte[] flippedIV = initialVectorBytes.Select(b => (byte)~b).ToArray();
```

```
        var cipher = new GcmBlockCipher(new AesEngine());
        var cipherParameters = new AeadParameters(new KeyParameter(aesKeyBytes

        // Encrypt the data
        cipher.Init(true, cipherParameters);
        byte[] encryptedDataBytes = new byte[cipher.GetOutputSize(dataToEncryp
        var offset = cipher.ProcessBytes(dataToEncrypt, 0, dataToEncrypt.Lengt
        cipher.DoFinal(encryptedDataBytes, offset);

        // Get the authentication tag
        byte[] authTag = new byte[TAG_LENGTH];
        Array.Copy(encryptedDataBytes, encryptedDataBytes.Length - TAG_LENGTH,

        // Concatenate encrypted data and auth tag, then return as base64
        byte[] encryptedResponse = new byte[encryptedDataBytes.Length - TAG_LE
        Array.Copy(encryptedDataBytes, 0, encryptedResponse, 0, encryptedDataB
        Array.Copy(authTag, 0, encryptedResponse, encryptedDataBytes.Length -
        return Convert.ToBase64String(encryptedResponse);
    }
}

// Helper class for providing a password to the PemReader
public class PasswordFinder : IPasswordFinder
{
    private readonly char[] _password;

    public PasswordFinder(string password)
    {
        _password = password.ToCharArray();
    }

    public char[] GetPassword()
    {
        return _password;
    }
}
```

Go Example

Here's a full code sample to handle a request with decryption/encryption in Go. Note that the response is sent as a plain text string.View the full project code on github.

```
package main
```

```
        "crypto/rand"
        "crypto/rsa"
        "crypto/sha256"
        "crypto/x509"
        "encoding/base64"
        "encoding/json"
        "encoding/pem"
        "errors"
        "fmt"
        "log"
        "os"

        "github.com/gin-gonic/gin"
)

const nonceSize = 16

type endpointPayload struct {
        EncryptedAESKey   string `json:"encrypted_aes_key"`
        EncryptedFlowData string `json:"encrypted_flow_data"`
        InitialVector     string `json:"initial_vector"`
}

type decryptionResult struct {
        DecryptedBody      map[string]interface{}
        AESKeyBytes        []byte
        InitialVectorBytes []byte
}

func main() {
        privateKey := os.Getenv("PRIVATE_KEY")
        passphrase := os.Getenv("PASSPHRASE")
        if privateKey == "" || passphrase == "" {
                log.Fatal("Environment variables 'PRIVATE_KEY' and 'PASSPHRASE
        }

        r := gin.Default()
        r.POST("/", func(c *gin.Context) {
                encryptedResponse, err := processRequest(c, privateKey, passph
                if err != nil {
                        log.Print(err)
                        c.String(500, "Internal Server Error")
                        return
                }
                // Return encrypted response as plain text
                c.String(200, encryptedResponse)
        })
        r.Run(":3000")
```

```
        var payload endpointPayload
        if err := c.ShouldBindJSON(&payload); err != nil {
                return "", err
        }

        // Decrypt the request data
        decrypted, err := decryptRequest(payload.EncryptedAESKey, payload.Encr
        if err != nil {
                return "", err
        }

        // Access decrypted fields
        action, ok := decrypted.DecryptedBody["action"].(string)
        if ok {
                fmt.Printf("Action: %s\n", action)
        }

        // Create a response object
        response := map[string]interface{}{
                "screen": "SCREEN_NAME",
                "data":   map[string]string{"some_key": "some_value"},
        }

        // Encrypt the response
        encryptedResponse, err := encryptResponse(response, decrypted.AESKeyBy
        if err != nil {
                return "", err
        }

        return encryptedResponse, nil
}

func decryptRequest(encryptedAESKey string, encryptedFlowData string, initialV
        // Parse the private key
        block, _ := pem.Decode([]byte(privatePem))
        if block == nil || !x509.IsEncryptedPEMBlock(block) {
                return decryptionResult{}, errors.New("invalid PEM format or n
        }

        decryptedKey, err := x509.DecryptPEMBlock(block, []byte(passphrase))
        if err != nil {
                return decryptionResult{}, err
        }

        privateKey, err := x509.ParsePKCS1PrivateKey(decryptedKey)
        if err != nil {
                return decryptionResult{}, err
        }
```

```
        aesKeyBytes, err := rsa.DecryptOAEP(sha256.New(), rand.Reader, private
        if err != nil {
                return decryptionResult{}, err
        }

        // Decrypt the Flow data
        initialVectorBytes, _ := base64.StdEncoding.DecodeString(initialVector
        flowDataBytes, _ := base64.StdEncoding.DecodeString(encryptedFlowData)

        blockCipher, err := aes.NewCipher(aesKeyBytes)
        if err != nil {
                return decryptionResult{}, err
        }

        gcm, err := cipher.NewGCMWithNonceSize(blockCipher, nonceSize)
        if err != nil {
                return decryptionResult{}, err
        }

        decryptedPlaintext, err := gcm.Open(nil, initialVectorBytes, flowDataB
        if err != nil {
                return decryptionResult{}, err
        }

        var decryptedBody map[string]interface{}
        if err := json.Unmarshal(decryptedPlaintext, &decryptedBody); err != n
                return decryptionResult{}, err
        }

        return decryptionResult{
                DecryptedBody:      decryptedBody,
                AESKeyBytes:        aesKeyBytes,
                InitialVectorBytes: initialVectorBytes,
        }, nil
}

func encryptResponse(response map[string]interface{}, aesKeyBytes, initialVect
        // Flip the initialization vector
        flippedIV := make([]byte, len(initialVectorBytes))
        for i, b := range initialVectorBytes {
                flippedIV[i] = ^b
        }

        // Encrypt the response
        jsonResponse, err := json.Marshal(response)
        if err != nil {
                return "", err
        }
```

```
                return "", err
        }

        gcm, err := cipher.NewGCMWithNonceSize(blockCipher, nonceSize)
        if err != nil {
                return "", err
        }

        encryptedData := gcm.Seal(nil, flippedIV, jsonResponse, nil)
        return base64.StdEncoding.EncodeToString(encryptedData), nil
}
```

NodeJS Script Demonstrating AES Key Encryption and Decryption

This NodeJS code sample aims to give an approximate example of how `encrypted_aes_key` field is calculated and how it can be decrypted.

```
// put private key in private_key.pem file in the same folder as this script
// run with: node <script-file-name>

import crypto from "crypto";
import fs from "fs";

const CLEAR_AES_KEY_STR = "<some-key-data>"
const PRIVATE_KEY_DATA = fs.readFileSync('private_key.pem', 'utf8');
const PUBLIC_KEY_DATA = fs.readFileSync('public_key.pem', 'utf8');

console.log("Clear key: " + CLEAR_AES_KEY_STR)

const encryptedAesKey = crypto.publicEncrypt(
  {
    key: PUBLIC_KEY_DATA,
    padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
    oaepHash: "sha256"
  }
  ,
  Buffer.from(CLEAR_AES_KEY_STR)
);

const encryptedAesKeyBase64 = Buffer.from(encryptedAesKey).toString('base64');

console.log("Encrypted base64 key: " + encryptedAesKeyBase64)

const decryptedAesKey = crypto.privateDecrypt(
  {
    key: crypto.createPrivateKey({
      key: PRIVATE_KEY_DATA,
      format: 'pem',
      type: 'pkcs1',//ignored if format is pem
      passphrase: '<passphrase>'
    }),
    padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
    oaepHash: "sha256",
  },
  Buffer.from(encryptedAesKeyBase64, "base64"),
);

console.log("Decrypted key: " + decryptedAesKey)
if (decryptedAesKey.toString() === CLEAR_AES_KEY_STR) {
  console.log("Success, keys match!")
} else {
  console.log("Failed, keys do not match!")
}
```