# Serverside Postprocessing using Callback Handlers



# The Scenario

In many situations, regardless if the envelope was sent via an API integration or manually sent from the eSignAnyWhere Web UI, a serverside postprocessing step should be imlpemented which is performing some activities after an envelope was completed, i.e. after all defined activities such as signing have been completed on all document(s) of a business case.

For that scenario, eSignAnyWhere allows to implement callback handlers, which are web services. These web services are invoked automatically, based on webhooks defined by the sender or the admin of an organization.

This story contains a summary of well known use cases where callback handlers help to solve common tasks, and describes best practices in implementing such.

We recommend to read the story "Retrieving the User API Token via OAuth" before continuing with this story, when you intend to cover situations where envelopes are sent via WebUI.

## Implementation Best-Practises

### Validate Input Parameters before processing

Keep in mind that the callback could have been fired from malicious sources. Therefore, validate all input parameters received via the callback. This includes:

- Is the action one of the expected ones?
  - Note that we might add other action codes in the future. Ignore those which should not be handled by your callback handler.
- Is the envelope ID in the proper format?
  - We are using the GUID format for envelope IDs
- Is the envelope in the expected status?
  - The "envelopeCallback" indicates that the envelope was completed. But don't trust just the occurance of an inbout callback. Validate via API calls if the envelopeId exists, and if the envelope is in the expected status.
  - Implement proper error handling, e.g. for cases where the envelope has been deleted before the callback handler was invoked.
  - As we recommend to implement asynchronous callback handling (see next topic), basic validation of the parameters should be done when noting down the callback - but another validation should be done before processing the event in the service job, as the envelope status or its existence might change.

### Asynchronous Callback Handling

Whenever a callback is received, the callback handler should just note down the event and return to eSignAnyWhere that the callback event was successfully noted down. But the event processing should be done asymetric, e.g. in a background service call.

If the implementation is done in the callback handler directly (in a synchronous processing), you may run in situations where the handling of a callback takes longer than the standard network timeouts configured on eSignAnyWhere side. When eSignAnyWhere receives a timeout due to longer ongoing processing activities in a callback handler, eSignAnyWhere is noting down the failure and starting a retry mechanism. In standard configurations, there are up to 30 retries of the callback. This may lead to situations where the callback processing of the same envelope, for the same event, is done up to 30 times or even higher with different system configuration. In case a callback handler is e.g. downloading signed documents from eSignAnyWhere, and uploading the documents to a document management system, an improper integration may result in storing the same document multiple times in the document management system.

Therefore, note down in a database which event was received, and for which envelope (or workstep, in case of WorkstepEventCallbacks). Let the callback handler return a HTTP 200 directly after the event was noted down in persistent storage. Implement a periodic serverside job which gets activated on a timed schedule, or on adding entries to the noted down events table. The serverside job then should asynchronously perform the necessary actions.

### Authentication retrieved via OAuth 2.0 Code Grant & note down the sender user

In some scenarios, e.g. those where the envelope is sent from WebUI, a callback handler implementation needs to deal with envelopes sent by different senders. While in scenarios where envelopes are sent via API, the sender's API key is well known at that time already (obtained via OAuth code or configuration), the user specific API key is unknown when sending via WebUI. We do therefore recommend to implement a BeforeSendRedirect page that implements the OAuth Code Grant flow, to obtain the consent from the sender to grant access via OAuth. The implementation then can retrieve, if unknown for the sender, the API key. We recommend to store the obtained API key per user in the database, under consideration of OWASP recommendations on how to persist passwords (e.g. encrypt them). Furthermore, note down in a different table which envelope (by envelopeId) was sent by which sender, as this information is necessary for the callback handler implementation.

In the callback handler implementation, do a lookup based on envelopeId which user was the sender; and then authenticate with the apiKey noted down for the sender. In case of an authentication error, trigger an error handling procedure to obtain a new apiKey (read below).

Alternative techniques such as using teams or using organization keys, are listed as "not recommended" for the following reasons:

- Having one team that contains all users and with a "technical user" as teamlead, so that the technical users as a teamlead has API access to all envelopes of the other senders, is not recommended because of privacy considerations, and in addition because of severe negative performance impact when using large teams with more than 20 or 30 users.
- The use of Organization Keys is not recommended for security considerations, and therefore the use of organization keys is by default disabled on newly generated organizations by settings that can be changed via administrative web only (on SaaS: Namirial staff only). Also, Organization Key authentication is not supported any longer with REST API v6.
- Using a "user determination service" which does the lookup per envelopeId directly on the database is not recommended as we don't guarantee to have a compatible DB schema with future product versions.

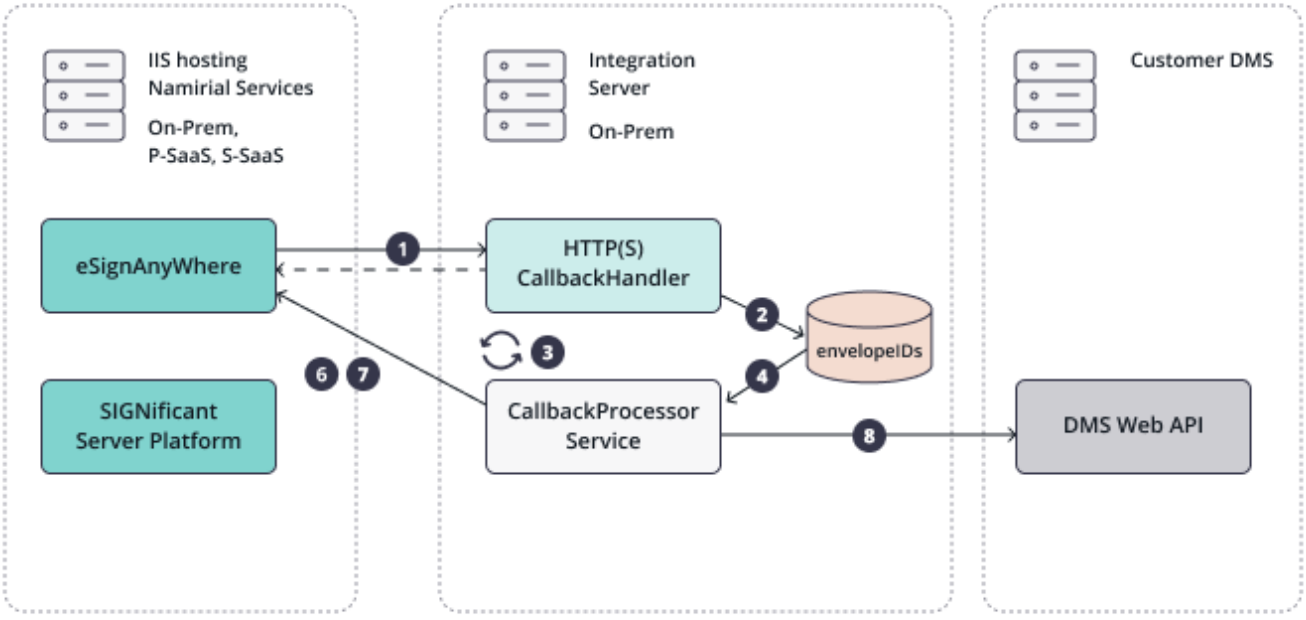
### Error Handling

Don't forget to think about proper error handling in your asynchronous callback handler. A callback handler is a serverside job which does not involve the user typically. But for situations where an error occured, think if the sender or an administrator needs to be involved. If a person has to be involved, as we are talking about automatic serverside processing, think about a proper way to trigger a human interaction - probably with some user specific or administrative UI of the callback handler, and a notification sent via e-mail. Typical scenarios you might consider in the implementation:

- API key is not valid any more, e.g. because a user revoked the permission for the OAuth a
- Unexpected technical errors e.g. in communication with the external system to be integrated
  For some of them the automatic retry might solve it, therefore note down how often the event occured to understand if there will be another retry anyways (e.g. if a remote system is not available).
  For those where all retries failed, a notification to someone checking system status and fixing the issue could be necessary.

## Example architecture diagram with call sequence

The diagram shows how a callback handler could push data to a document management system (DMS). Any other integration system that needs to process signed documents could be in place of the "Customer DMS" instead.




Following steps describe the implementation in detail:

| Number in the figure | Description |
|---|---|
| 1 | eSignAnyWhere calls your CallbackHandler implementation when the envelope is finished (completed or cancelled). |
| 2 | Your CallbackHandler stores the envelope ID in a persistent storage (disk, database) and replies with HTTP 200 on the HTTP request (1). |

| | |
|---|---|
| 3 | A service implementation should permanently process the envelopes noted-down for processing. |
| 4 | A service implementation will detect the noted-down envelope ID and process it as step by step queue processing. |
| (5) | (removed) |
| 6 | It should call the REST API call GET envelope/{envelopeId} to receive details about the envelope, including metaDataXml which was set by the DMS tagging application, and receive document IDs for the relevant documents. |
| 7 | The documents, which are typically the signed PDFs, the Audit Trail PDFs, and in addition the legal documents (such as the CA22D certificate request form for disposable certificates) should be downloaded form eSignAnyWhere via API. |
| 8 | After that, the application should upload the documents to the external DMS via the DMS API according to the requirements of the DMS; and use the tagging retrieved from the metadata. |

Once completed, the callback handler implementation (i.e. the asynchronous service) should delete the envelope from eSignAnyWhere when it is not used in eSignAnyWhere any longer and after the document, the audit trail and other legal evidence documents have been successfully transferred to a permanent storage outside eSignAnyWhere.

Alternatively, a data retention job can be configured on the organization to perform an automatic cleanup. But note that the retention job would not identify /stop when the transfer to the permanent storage failed.