Media Pool

Description of Webhook Push API

Version 6.6

EN

3 February 2020

Table of Contents

Copyright

Rüppurrer Straße 1, 76137 Karlsruhe (Germany), www.brandmaker.com

Your feedback is important to us!

We would be grateful to be notified of any errors you may discover. Just send us an e-mail to documentation@brandmaker.com.

1 Introduction and Objectives

The BrandMaker Media Pool is increasingly used by its users as a central repository for all types of assets that are also to be used outside the company. The Media Pool is intended to assume the role of a central application that controls the distribution of assets. It is therefore necessary for the Media Pool to provide powerful APIs that enable users to synchronize and keep stored content and third-party applications up to date.

1.1 Task Definition

In order to deliver assets in the Media Pool to third-party applications, third-party applications must be able to be notified of changes to relevant assets.

In order to load these assets from the Media Pool, the Media Pool must have APIs for searching and loading metadata and for transferring binary data. This allows users to deploy and keep assets from the Media Pool in their own applications (e.g., CMS, PIM, CRM). In order to achieve the set goals, the Media Pool already provides extensive REST APIs and SOAP APIs. These APIs aren't described in more detail below. Please refer to the existing documentation.

New and described in this document are the so-called Webhooks, which can inform third-party applications about modifications to assets in the Media Pool. The basic idea is the following: Third-party applications is notified of changes to assets in near real time, rather than having to perform cyclical scans of the entire inventory to identify such changes.

A very comprehensive description of how Webhooks work can be found at https://requestbin.com/blog/working-with-webhooks/ and https://en.wikipedia.org/wiki/Webhook.

For this purpose, the third-party applications should register at the Media Pool. This registration consists of storing a URL (Webhook) that is called by the BrandMaker system as soon as relevant changes have been made to the asset.

All actions performed on an asset by users generate an event. When registering the webhook, the administrator can select one or more of these events. Please note the flow chart Figure 1 - Functional overall structure in chapter 2.

1.2 Boundary Conditions

It is assumed that the reader or developer on the customer side is familiar with technologies such as http, REST and the programming of web services.

Listed example listings are exemplary created in JAVA, but there is no constraint and no preference for any programming language.

Furthermore, knowledge of the application and configuration of the BrandMaker Media Pool is required. The developer needs administrator access to the Media Pool to be able to make the necessary settings.

In order to retrieve and save data from the Media Pool automatically using a third-party system (e.g. CMS), knowledge of the corresponding development environments and APIs of the third-party system is required. This knowledge is also not covered in this documentation. The third-party system is always represented as a black box in this context.

1.3 Defining the Context

The offered interfaces are used for near real-time synchronization of the data in the Media Pool with third-party applications. Other purposes are not supported.

Technical context

From the point of view of the Media Pool, delivery to a third-party system is referred to as publishing. When the user specifies that an asset is to be published, the asset becomes available for transfer to a third-party system.

Assets can be published to different systems in parallel. Such an external system is referred to as a channel. A channel can be your own CMS, a social media platform, a blog, or even an external system for print preparation.

In order for the third-party application to be able to download the asset in the correct format, one of the available output formats must be specified per channel. For this purpose, a corresponding rendering scheme must be defined in the Media Pool and stored with the channel.

The publishing process therefore includes the following points:

- the selection of the asset
- the definition of one or more channels
- the definition of the rendering scheme
- the determination of the publication period

The following requirements must be met for third-party applications to be notified via webhook of changes to an asset:

- The asset has been published.
  - A channel is defined.
  - A publication period is defined for the asset and the change to the asset takes place within the publication period.
  - A rendering scheme is specified that contains an output format for the asset's file format.
- A third party application is defined for the channel.
  - A valid URL of the third-party application has been filled in.
  - The events for which the third-party application is to be notified are defined.

With these settings, the webhook sends notifications to the third-party application as soon as one of the changes registered as an event is made to the asset.

The third-party application that receives the notification can then respond by, for example, loading the new version of the asset or resetting the asset's title.

2 Structure

A complete system consisting of Media Pool, third-party application and its coupling as well as the necessary interfaces can look like the following example:

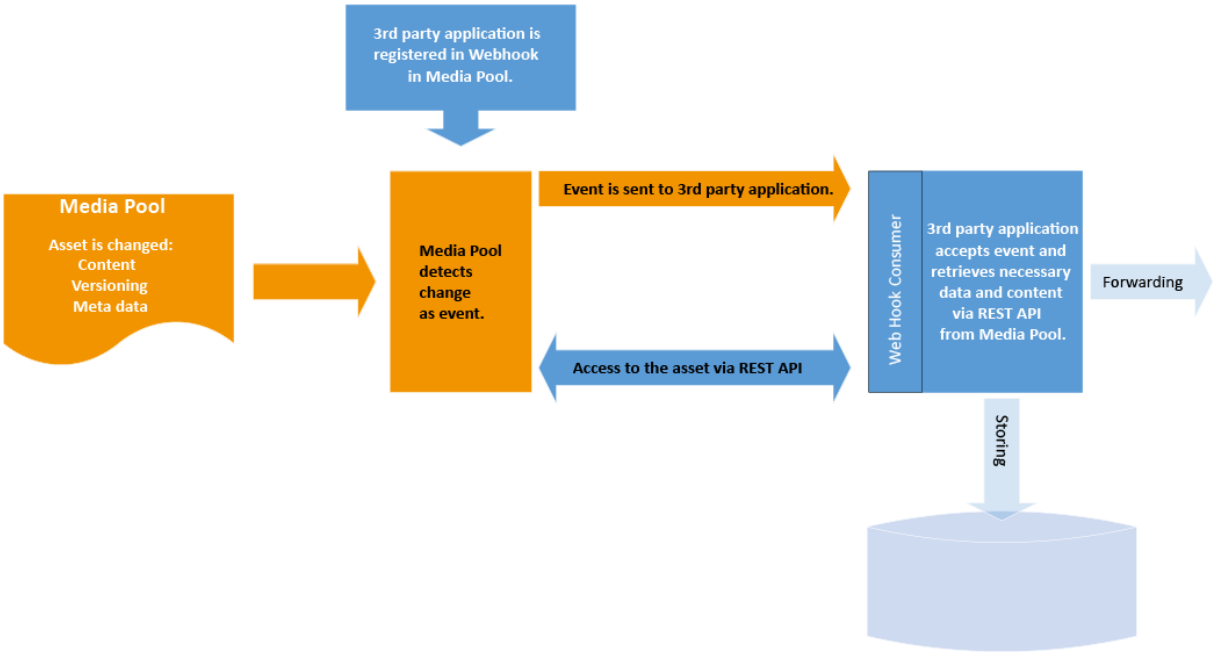


Figure 1 - Functional overall structure

Media Pool	BrandMaker Media Pool Instance
Third party application	The third-party application that is to receive the released, published assets from the Media Pool
Media Pool REST API	API to retrieve data from the Media Pool
Webhook-Consumer	The REST endpoint to create within the third-party application that receives the notifications

3 Description of the Components and Their Function

3.1 Media Pool

The Media Pool is the active component, which as the leading system can supply third-party applications with data and content. To do this, the Media Pool informs the third-party application via webhook about events that have occurred in the system. The following events can be assigned to a webhook.

3.1.1 List of Asset Related Events

The following events report a direct change to an asset:

#	Event	Data submitted	Description
4	PUBLISHED	ID, channel ID, rendering scheme, startDate, endDate	An asset has been published to one of the channels. The event fires separateley for each affected channel.
5	PUBLISHING_START	ID, channel ID, rendering scheme, startDate, endDate	A publishing time has been reached. The event fires as soon as the asset is due to be published on a particular channel.
6	PUBLISHING_END	ID, channel ID, startDate, endDate	A publishing end date has been reached. The event fires as soon as the asset gets depublished on a particular channel.
7	DEPUBLISHED	ID, channel ID, startDate, endDate	The asset is manually depublished by the user. This is not the same as "PUBLISHING_END" as in this case the asset is still in a published state but the end of the "To" date has been reached
8	METADATA_CHANGED	ID	Event fires if ANY metadata has been changed. This includes all custom attributes (aka "free text fields"). Except: Versions, Variants and related assets. This also does NOT INCLUDE changes to the category tree associations. These are treated separately

#	Event	Data submitted	Description
9	VERSION_ADDED	ID, Version #, channel ID, rendering scheme	Fires if a new version is added
10	VERSION_DELETED	ID, Version #	Fires, if any version is deleted
11	VERSION_OFFICIAL	ID, Version #, channel ID, rendering scheme	Fires, once a version is set to official
12	VERSION_UNOFFICIAL	ID, Version #	Fires, once a version is set to unofficial
23	SYNCHRONIZE		Send for full sync on manual triggering the web hook. Cannot be selected as automatic triggering event
24	TEST		Test event which only checks whether the endpoint is listening. Cannot be selected as automatic triggering event

The numbering of the events is not consecutive. Further events will be added in later versions.

The most importants events are `PUBLISHED`, `DEPUBLISHED`, `PUBLISHING_START`, `PUBLISHING_END`, `ASSET_REMOVED`. These events monitor the publication of assets.

Please note that the data of an asset is not transmitted to the Webhook (content), only the information about what has changed on an asset. The recipient of the webhook, the consumer, must then decide how this information is processed and how necessary data and content is loaded via the REST API provided by the Media Pool.

The data transmitted in the webhook is sent to the REST endpoint of the consumer in JSON notation using the POST method.

The Media pool assumes that no authentication takes place in the webhook. To ensure that the request was sent by the correct instance, the data is signed with the private key of the BrandMaker system and can be verified using the public key of the BrandMaker system. The public key can be requested via an API of the BrandMaker system, please refer to the administration manual of the BrandMaker system. Since the source URL is also contained in the signed data, a positive validation of the signature verifies the source system.

3.1.2 List of Events Showing Changes in the Categories

General category changes also affect the metadata of an asset. However, since no changes have been made to the assets themselves in the event of a change in the categories, events are not sent for the affected assets, but for the corresponding categories.

#	Event	Data submitted	Description
25	CATEGORY_MOVE	ID, Source, Target Category ID	**Attention:** This is actually not a change in the asset, but in the category tree: a sub-category is moved from one parent to another parent. This affects all assets assigned to this sub-category! For sync purposes, where the client is as well synchronizing the category tree, it's important to know about such *implicit moves* of assets.
26	TREE_CHANGED	no data	If the above "CATEGORY_..." events cannot be fired due to certain implementation restrictions, the "TREE_CHANGED" event should be used instead whenever a change to the category tree is saved. This will signal the client that it has to sync the entire categroy tree. The current associations may as well need to be resynchronized, as for instance a category has been removed and the according assets have changed.

3.2 Webhook Consumer

To process the data sent by the Media Pool via webhook, the third-party application must provide a REST endpoint whose URL the user must register and activate in the BrandMaker system.

Depending on the event that has occurred, different data objects are sent that contain the necessary information for further processing of the event.

> **Note**
>
> Meta data or content of assets are **not** sent!

The URL of the REST endpoint is called by the Media Pool immediately after the corresponding transaction within the Media Pool is completed. Depending on the system load and the number of events that arrive and are to be transferred, this can take a few seconds.

If an error occurs at the REST end point of the consumer, the Media Pool attempts to resend the event. The repetition rate can be adjusted by the BrandMaker support, please contact the BrandMaker support if necessary. You specify the time interval between two attempts when you register the webhook in Administration.

If all retries fail, the Webhook is locked and no further notifications are sent. In this case, manual intervention is necessary. This is indicated via the status display of the webhook in the BrandMaker system under > *Administration* > *Media Pool* > *Webhooks*.

> **Note**
>
> All HTTP response codes outside the range 20x are considered to be faulty. By default, the webhook expects an HTTP response status 202 - Accepted in case of success. Please refer to the implementation recommendations in chapter 3.2.2. Please note that a series of successive errors of the consumer may cause the webhook to be automatically deactivated and must be manually reactivated.

3.2.1 Structure of Transmitted Data

In principle, the data sent to the webhook has the following structure:

```
{
     "data": "{\"customerId\": \"una-nho-eie\",\"systemId\": \"821-574-160\",\"baseUrl\":
\"https://is-dev2.brandmaker.com/\",\"events\": [ { \"assetId\": \"3467\", \"eventData\":
[{ \"channelId\": \"PUBLIC_LINKS\", \"startDate\": null, \"endDate\": null,
\"renderingScheme\": 856}], \"eventType\": \"PUBLISHED\", \"eventTime\": 1552667068052
}]}",
     "signature": "sdlk.................fhsfgjhg"
}
```

- `data` – contains the actual event data JSON object as a string
- `signature` – contains the signature of the data element

In the further course of the document only the decoded part of the data element is considered:

```
{
 "customerId": "id",
 "systemId": "id",
 "baseUrl": "url",
 "events": [{
     "assetId": "id",
     "eventData": [{
         "channelId": "PUBLIC_LINKS",
         "startDate": null,
         "endDate": null,
         "renderingScheme": "id"
     }],
     "eventType": "PUBLISHED",
     "eventTime": "UTC"
 }]
}
```

To optimize throughput, several events occurring in succession are combined in the Media Pool in one call. Therefore, the `events` element contains an array of the events that have occurred.

The `eventData` element contains different data depending on the type of event, see the table of events.

Since the user can publish an asset on several different channels, this element also contains an array with one data object per addressed channel.

Content of JSON object

Element	Description
`customerId`	Unique customer number in format 111-111-111 (alphanumeric)
`systemID`	Unique System ID in format abc-def-ghi (alphanumeric)
`baseUrl`	Web address of system for API calls by 3rd party application
`assetId`	Asset ID of the affected asset(s) (without prefixed M-)

Element	Description
`channelId`	ID pf publication channel. By default Media Pool has two fixed channels: SHARING and PUBLIC_LINKS. See the Media Pool Administration Manual.
`renderingScheme`	ID of the desired output format of the asset. This is either preset in the channel or specified for each channel when publishing.
`eventType`	Event, see chapter 3.1
`eventTime`	Effective time of the event as UTC time stamp

3.2.2 Implementation Recommendations

In principle, the consumer to be created must have good performance and respond relatively quickly, since the Media Pool only waits a limited time for the response and otherwise classifies the call as faulty (timeout).

Since several requests can arrive at the same time and these can also arrive for several events, we strongly recommend asynchronous processing in the following schematic steps:

1. Acceptance of the request
2. Formal validation of the data
    a. Verification of the signature
    b. Formatting the Event Object
    c. Entry in internal queue
3. Reply to Media Pool

Tests have shown that in this way a response is in principle possible within 50-100 milliseconds (JAVA, JMS, Active-MQ).

An internal queue consumer can then record the actual processing of the events.

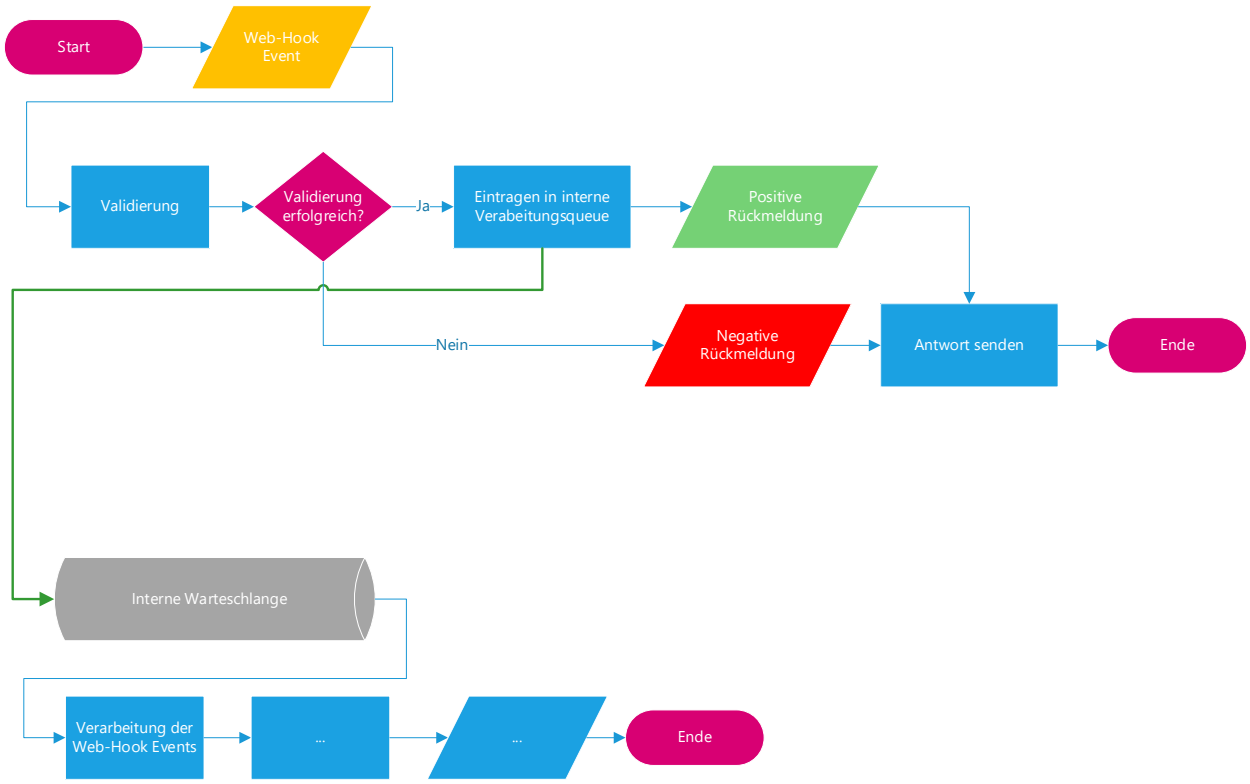


Figure 1 - Principle sketch webhook consumer

4 Relevant REST APIs for synchronization

To retrieve the effective data of an asset from the Media Pool, the Media Pool provides a comprehensive REST API.

All API endpoints require authentication. This is currently done as Basic Authentication. BrandMaker recommends setting up access to the APIs of a non-personal, technical user whose password is always valid.

Please note that in one of the next versions of BrandMaker, the authentication of the APIs will be changed centrally to the oAuth2 standard in connection with the BrandMaker Fusion feature set.

The following REST APIs are of interest in connection with this document:

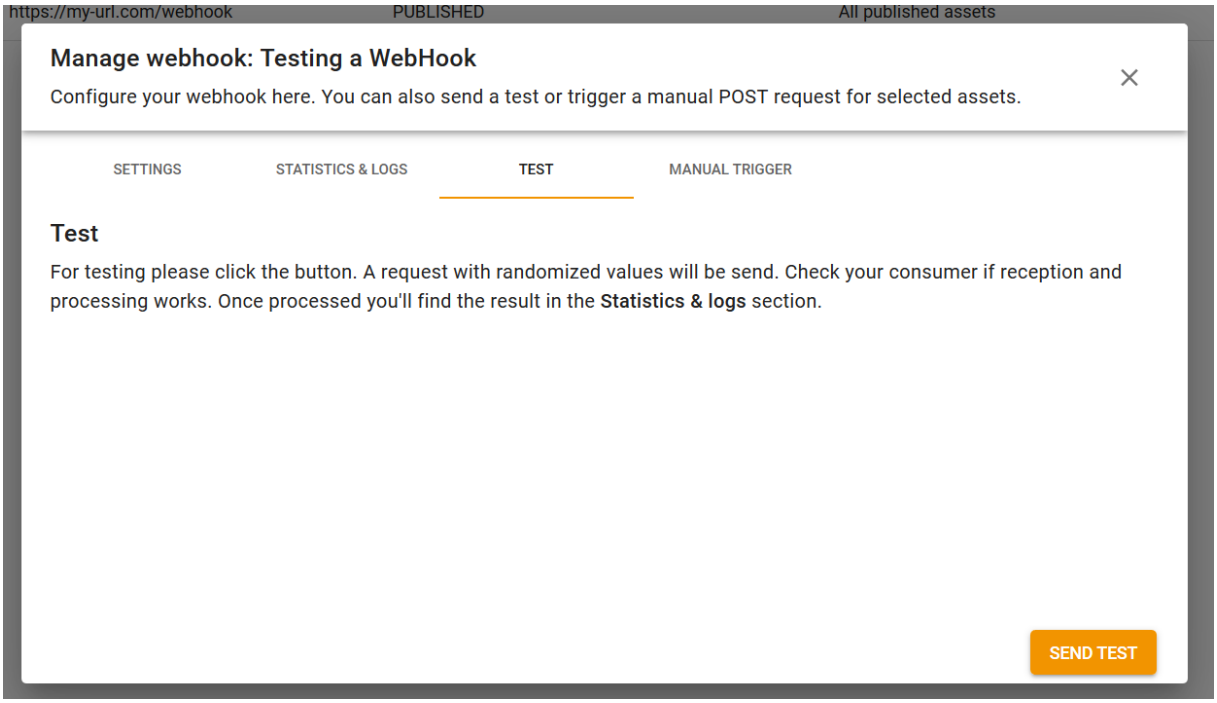
REST Resource	Description
AssetSearchRestService	Search for assets. This is also recommended for pulling required metadata for a specific Asset ID, since the scope of the returned data can be configured via the call.
FileGenerationTaskRestService DownloadRestService	These two resources create the appropriate rendering for the contents of an asset. Note that rendering an asset can be time-consuming. For this reason, a task is first created (*FileGenerationTask*) and this can be queried with the *DownloadRestService* to see if the asset is rendered in the desired format. The ID of the requested rendering scheme serves as a basis.
FileResourceRestService	Provides detailed information about an asset's stored file format.
AssetResourceVersionRestService	Detailed information about the versioning of an asset. The resource *FileGenerationTaskRestService* expects information about the version of the asset to be delivered, which can be determined here.

Detailed information on the individual resources of the REST API as well as further endpoints can be found within the BrandMaker system under > *Administration* > *System Information* > *API Descriptions*.

5 Test of Consumer

During creation you can test your new webhook consumer via the Media Pool. To do this, create a webhook with the valid address of the consumer. Note that this address must be resolved and must be accessible from outside via the internet. This is usually not the case for your local computer.

Go to the *Test* tab in the administration of the webhook and click on the *Send Test* button:



A corresponding event TEST (see chapter 3.1) without user data is sent to your consumer and the return status is stored in the log.

Please note that in the Media Pool, the events are also generated asynchronously. It is therefore possible that manually triggered events (test, manual trigger) only actually arrive at the consumer after a few seconds, depending on the system load.

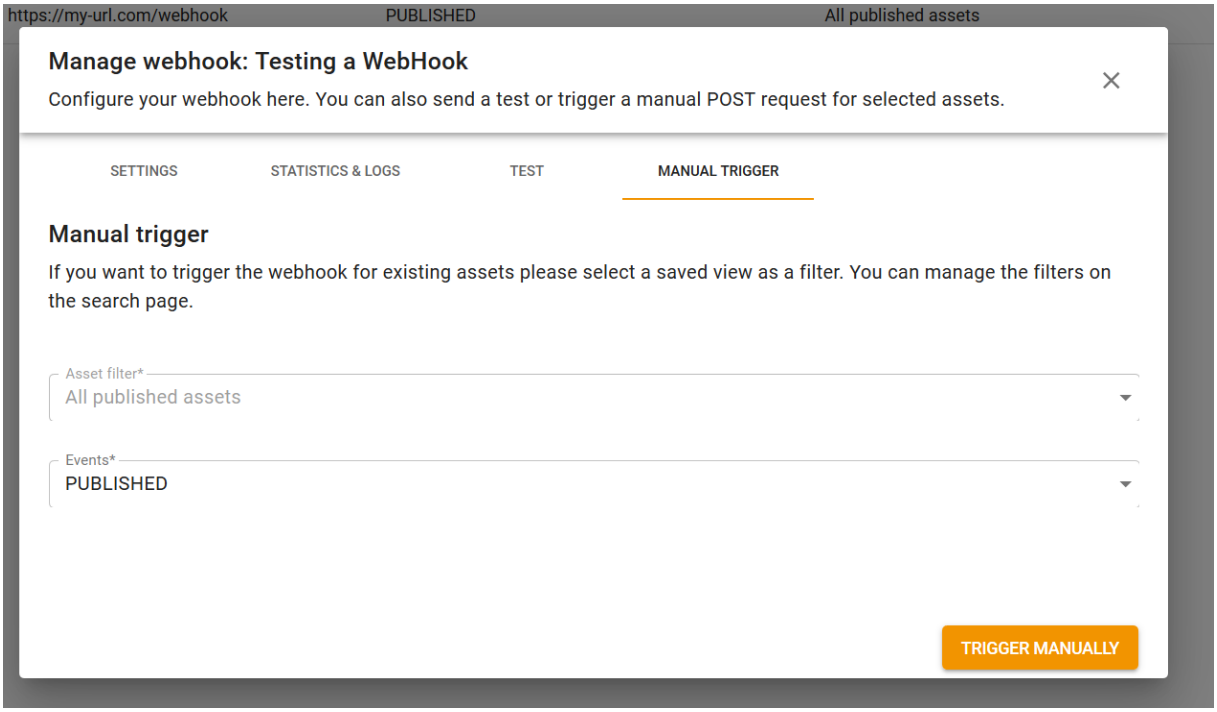
6 Loading an Existing Data Pool

When your webhook is complete, it receives events for modifications to the assets according to the registration.

However, this does not apply to assets published before the Webhook was created. To avoid having to republish them all, use the `SYNCHRONIZE` event, which is sent to all assets that meet the conditions.

Please note that this sends the corresponding events to your webhook for all assets ever published and currently valid in the system. The events are sent in clusters of multiple assets at once (200 per request). Nevertheless, a high system load is to be expected.

To do this, go to the *Manual Trigger* tab in the Webhook's administration dialog:



7 Possible Error Sources

Im Folgenden werden einige mögliche Symptome und deren mögliche Ursachen als Hilfe bei der Implementierung aufgelistet:

Indication	Possible cause and solution
The request is not received on the target system.	• The Webhook contains an invalid or incorrect URL. *Solution:* If necessary, correct the URL entered in the BrandMaker system under > *Administration* > *Media Pool* > *Webhooks* for this webhook. • The BrandMaker system cannot reach the third-party application with the webhook. *Solution:* Make sure that the third-party application with the webhook is accessible via the Internet. If necessary, check the DNS resolution and the routing.
The request is received, but returns an http status unequal 20x.	• The format of the data does not correspond to the format expected by the Webhook or there is a fundamental implementation error. Solution: Check and correct your implementation if necessary.
An event is not sent to the webhook.	• The webhook has not subscribed to the corresponding event. *Solution:* Check and correct the webhook settings under > *Administration* > *Media Pool* > *Webhooks*. • The corresponding asset is not published. Please note that only published assets are processed via this API. *Solution:* Search for the asset in the Media Pool. Check and correct the publication of the asset if necessary.
The webhook is automatically deactivated.	• Your consumer delivers too many errors in succession. *Solution:* Check the stability of your implementation, especially in the processing of the various events enabled in the registry.
Media Pool reports that timeouts occur.	• Processing within the Webhook consumer takes too long. *Solution:* Check the Webhook consumer and follow the instructions in chapter 3.2.2.

Indication	Possible cause and solution
Modifications to the assets are not sent to the Webhook, even though the corresponding events are set up and the asset is published.	• The specified rendering scheme has changed and does not contain a valid output format for the file format of the asset. *Solution:* Change the rendering scheme specified for the publication under > *Administration* > *Media Pool* > *Download* > *Rendering Scheme*. Alternatively, select a different rendering scheme for the asset's publication that includes a suitable output format. • The specified rendering scheme was deleted. *Solution:* Select a different rendering scheme for the asset's publication that includes a suitable output format.
An asset is automatically reported as DEPUBLISHED before its expiration date and without user interaction.	• The asset has been moved to the VDB recycle bin. In this case the publication of the asset will be automatically stopped from version 6.6 on.
The webhook consumer cannot access the Media Pool API.	• No physical connection can be established. *Solution:* Check the network connections of your server and the BrandMaker system.
The Media Pool API returns an HTTP status 40x.	• The technical user with which the API authenticates itself to the BrandMaker system is invalid or does not have sufficient rights. *Solution:* Check the user account of the technical user under > Administration > Users & Groups > Users. • Information about the requested resource (asset, rendering scheme, file information, ...) is not available. *Solution:* Please refer to the corresponding API documentation.

For further possible causes of errors in connection with the Media Pool API, please refer to the corresponding documentation of the individual REST endpoints used.

8 Examples

8.1 Webhook Rest Endpoint

Below is an example implementation of a REST endpoint based on Java 8, Apache Sling 11, and JAX-R:

```
/**
 *
 * <p>Web Hook which retrieves events generated by Media Pool in order to sync assets between Media Pool and WebCache.
 * <p>This service is not doing anything else than making formal checks on the request and then queue the request to the importer queue!
 * <p>As this API does not require any login, it is crucial to validate the request signature!
 *
 * @author axel.amthor@brandmaker.com
 * @copyright BrandMaker GmbH, Karlsruhe, 2019
 */
@Service(value={MediaPoolWebHook.class}) // service on interface in order to hide class specific implementations!
@Component(
        label = "MediaPool WebHook Service",
        description = "Captures requests with Media Pool Sync Events",
        metatype = true
)
@Properties({
   @Property(
           label="WebHook job creation",
           description="Whether sync jobs should be created if an event is passed in. Defaults to false (!)",
           name="MediaPoolWebHook.Jobcreation.active",
           boolValue = false
           ),
   @Property(
           label="WebHook service active",
           description="Whether WebHook Service is activated. Defaults to false (!)",
           name="MediaPoolWebHook.Service.active",
           boolValue = false
           )
})
public class MediaPoolWebHookImpl implements MediaPoolWebHook
{

   private final static boolean DEBUG = true;

   /** The Constant LOGGER. */
    private final static Logger LOGGER = LoggerFactory.getLogger(MediaPoolWebHook.class);

    /** Whether this service should be activated or not */
    private boolean serviceActive = false;

    /** false means requests are accepted but not queued */
    private boolean queueActivated = false;

   @Reference
   ReaderService readerService;

   @Reference
   WebCacheTenantProvider tenantProvider;

   @Reference
   ResourceResolverHelper resourceResolverHelper;

   @Reference
   private JobManager jobManager;

   /**
    * Activate this service component and init global stuff.
    *
    * @param bundleContext
    * @param properties
    */
```

```
61     @Activate
62     public void activate(final BundleContext bundleContext, final Map<String, Object>
    properties)
63     {
64        serviceActive  =
    PropertiesUtil.toBoolean(properties.get("MediaPoolWebHook.Service.active"), serviceActive);
65        queueActivated  =
    PropertiesUtil.toBoolean(properties.get("MediaPoolWebHook.Jobcreation.active"),
    queueActivated);
66
67        LOGGER.info(WebCacheUtils.MARK, "MediaPool WebHook Service started");
68     }
69
70     /* (non-Javadoc)
71      * @see
    com.brandmaker.webcache.core.asset.jaxrs.mediapool.MediaPoolWebHook#mpWebHook(javax.servlet
    .http.HttpServletRequest, javax.servlet.http.HttpServletResponse, java.lang.String)
72      */
73     @Override
74     public Response mpWebHook(HttpServletRequest request, HttpServletResponse response,
    String requestBody)
75     {
76        long start = System.currentTimeMillis();
77
78        if ( !serviceActive ) {
79            LOGGER.info(WebCacheUtils.MARK, "Service inactive" );
80            return Response.status(Response.Status.NOT_IMPLEMENTED)
81                    .entity( "{\"error\": \"This service is inactive. Please check WebCache
    Configuration.\" } " )
82                    .type(MediaType.APPLICATION_JSON).encoding("UTF-8")
83                    .build();
84        }
85
86        LOGGER.info(WebCacheUtils.MARK, "Start processing webhook request from " +
    request.getHeader(WebCacheConstants.EFFECTIVE_CLIENTIP_ADDRESS_HEADER) );
87
88        JSONObject eventObject = new JSONObject();
89        JSONArray responseArray = new JSONArray();
90        String[] copyProps = { MediaPoolEvent.PROP CUSTOMERID, MediaPoolEvent.PROP SYSTEMID,
    MediaPoolEvent.PROP_BASEURL };
91
92        try
93        {
94            JSONObject requestObject = new JSONObject(requestBody);
95
96            String data = requestObject.getString("data");
97            String signature = requestObject.getString("signature");
98
99            /*
100            * validate signature
101            */
102           // implementation specific!
103
104           /*
105            * parse data property and parse the inner structure as JSON
106            */
107           JSONObject dataObject = new JSONObject(data);
108           JSONArray eventArray = dataObject.getJSONArray("events");
109
110           if ( DEBUG ) LOGGER.info(WebCacheUtils.MARK, "decoded data: " +
    dataObject.toString(4));
111           /*
112            * process event array
113            */
114           for ( int n = 0; n < eventArray.length(); n++ )
115           {
116               eventObject = eventArray.getJSONObject(n);
117
118               /*
119                * these props need to go into each event, as within the subsequent queue,
    there is no "batch" but single events
120                */
121               for ( String prop : copyProps ) {
122                   if ( dataObject.has(prop) )
123                       eventObject.put(prop, dataObject.getString(prop));
124               }
125
126               /*
```

```
127                 * validate event data
128                 */
129                MediaPoolEvent event = requestValidation(eventObject);
130
131                if ( event != null )
132                {
133                     /*
134                      * Test event, we just respond with a 202 and event data as content,
135                      * no futher processing!
136                      */
137                     if ( event.getEvent() == MediaPoolWebHookEvents.Event.TEST ) {
138                         responseArray.put(event.toJson());
139                     }
140                     else {
141
142                          /*
143                           * if the event is not dedicated to one of the two WebCache
    Channels, it must not be queued
144                           * if not, we just guzzle up the event, send an "accepted" back but
    actually are doing nothing
145                           */
146                          if ( !event.mustHaveChannel() || event.isWebCacheChannel() )
147                          {
148                             /*
149                              * Check whether we know the tenant by system and customer ID
150                              */
151                             WebCacheTenant tenant =
    tenantProvider.getTenantBySystemAndCustomerId(event.getSystemId(), event.getCustomerId() );
152
153                             if ( tenant == null ) {
154                                    eventObject.put("error",  "System ID and/or Customer ID
    unknown.");
155                                    throw new MediaPoolSyncJobException("No tenant found.");
156                             }
157                             event.setTenantId(tenant.getAccountId());
158
159                             if ( DEBUG ) LOGGER.info(WebCacheUtils.MARK, "MP Sync Request: "
    + event.toJson().toString(4));
160
161                             // this user is running the import later
162                             Principal userPrincipal = request.getUserPrincipal();
163                             if ( userPrincipal != null )
164                                    event.setUser(userPrincipal.getName());
165                             else
166                                    event.setUser("anonymous");
167
168                             if ( queueActivated || eventObject.optBoolean("BenGurion") )
169                                    addMediaPoolSyncJob(event);
170                             else
171                                    LOGGER.info(WebCacheUtils.MARK, "MP Sync Request but
    queueing deactivated");
172
173                             responseArray.put(event.toJson());
174                          }
175                          else {
176
177                             eventObject.put("error",  "not a webcache channel: " +
    event.getChannelsFromPayload() );
178                             responseArray.put(eventObject);
179
180                             LOGGER.info(WebCacheUtils.MARK, "not a webcache channel: " +
    event.getChannelsFromPayload() );
181                          }
182                     }
183                }
184                else {
185                     return Response.status(Response.Status.BAD_REQUEST)
186                             .entity( "{\"error\": \"Problems parsing event data.\" } " )
187                             .type(MediaType.APPLICATION_JSON).encoding("UTF-8")
188                             .build();
189                }
190           }
191
192           return Response.status(Response.Status.ACCEPTED)
193                    .entity( responseArray.toString(4) )
194                    .type(MediaType.APPLICATION_JSON).encoding("UTF-8")
195                    .build();
196
```

```
197         }
198         catch (JSONException e)
199         {
200             LOGGER.error("A JSON error occured", e);
201             LOGGER.info(WebCacheUtils.MARK, "(1) Invalid MP Sync Request: " + requestBody );
202             return Response.status(Response.Status.BAD_REQUEST)
203                     .entity( "{\"error\": \"invalid JSON in request.\" } " )
204                     .type(MediaType.APPLICATION_JSON).encoding("UTF-8")
205                     .build();
206         }
207         catch (MediaPoolSyncJobException e)
208         {
209             LOGGER.error("An error occured", e);
210             LOGGER.info(WebCacheUtils.MARK, "(2) Invalid MP Sync Request: " + requestBody );
211             try
212             {
213                 return Response.status(Response.Status.BAD_REQUEST)
214                         .entity( eventObject.toString(4) )
215                         .type(MediaType.APPLICATION_JSON).encoding("UTF-8")
216                         .build();
217             }
218             catch (JSONException e1)
219             {
220                 LOGGER.error("A JSON error occured", e1);
221             };
222         }
223         catch (Exception e)
224         {
225             LOGGER.error("A general error occured", e);
226             LOGGER.info(WebCacheUtils.MARK, "(4) Invalid MP Sync Request: " + requestBody );
227             return Response.status(Response.Status.BAD_REQUEST)
228                     .entity( "{\"error\": \"Request not processed.\" } " )
229                     .type(MediaType.APPLICATION JSON).encoding("UTF-8")
230                     .build();
231         }
232         finally
233         {
234             LOGGER.info(WebCacheUtils.MARK, "Finished processing webhook request from " +
    request.getHeader(WebCacheConstants.EFFECTIVE CLIENTIP ADDRESS HEADER) +
235                     " in " + (System.currentTimeMillis() - start) + " msec");
236         }
237
238         LOGGER.info(WebCacheUtils.MARK, "(3) Invalid MP Sync Request: " + requestBody );
239         return Response.status(Response.Status.BAD_REQUEST)
240                 .entity( "{\"error\": \"Request not processed.\" } " )
241                 .type(MediaType.APPLICATION_JSON).encoding("UTF-8")
242                 .build();
243
244
245     }
246
247
248     /**
249      * Validates the request data and returns an event Object if valid, otherwise null.
250      * Error messages are put back into therequest object as "error": "message..."
251      *
252      * @param requestObject
253      * @return true if data is valid, false otherwise
254      * @throws JSONException
255      *
256      */
257     private MediaPoolEvent requestValidation(JSONObject requestObject) throws JSONException,
    MediaPoolSyncJobException, Exception
258     {
259         MediaPoolEvent event = null;
260         try
261         {
262             event = new MediaPoolEvent(requestObject);
263         }
264         catch (Exception e)
265         {
266             LOGGER.error(WebCacheUtils.MARK, requestObject.toString(4) );
267             LOGGER.error(WebCacheUtils.MARK, e.getMessage(), e);
268             requestObject.put("error", e.getMessage());
269             throw e;
270         }
271
272         return event;
```

```
273     }
274
275
276     /**
277      * Create an importer Job to sync the assigned information
278      * @param asset
279      * @param request
280      * @throws MediaPoolSyncJobException
281      * @throws Exception
282      */
283     private void addMediaPoolSyncJob(MediaPoolEvent event) throws MediaPoolSyncJobException
284     {
285          // add necessary objects for conversion:
286          final Map<String, Object> eventMap = event.toMap();
287          final HashMap<String, Object> props = new HashMap<String, Object>();
288          props.put(PROP MEDIAPOOLEVENT,  eventMap);
289
290          Job job = jobManager.createJob(IMPORTER_JOB_TOPIC).properties(props).add();
291
292          if ( job == null ) {
293            LOGGER.error("No Job created ");
294            throw new MediaPoolSyncJobException("No job created");
295          }
296          else
297            LOGGER.info("Job " + job.getId() + " created in: " + job.getQueueName() );
298     }
```