**MePub Ingestion Overview by Jacob**

June 6, 2023, 3:32PM

30m 54s

 **Jacob Kissel** 0:07  
I know that I have to do something weird to.

 **Kush Mishra** joined the meeting

 **Jacob Kissel** 0:12  
Make sure that it shares the right way.

 **Aravind Siddoju** joined the meeting

 **Achal Shah** 0:17  
We can see your screen or the PowerPoint.

 **Jacob Kissel** 0:20  
OK. Ah.  
Out here.  
Nope, not this one.  
OK.  
Yep, so this is gonna be kind of, uh, barely basic overview.  
Apparently I'm presenter name and I'm actually Jacob.  
We're gonna.  
I'm gonna just talk about ingestion and kind of the tenants of ingestion for the media publishing team and kind of the ideas we try to follow across our different media types and then how we implement these across a couple examples.  
And then a brief look at the code just to illustrate the shared paradigm.  
As we use umm, I think I should be able to see if anyone has her hand up.  
But if I can't, then definitely feel free to interrupt and I'll try to answer any questions I have answers to and then if not, hopefully maybe a couple other people are here from me.  
Pub that can help answer about different media types.  
For example umm, so the first thing is what kind of tenants we follow when we talk about ingestion O for media publishing there are a couple different request types in of course create is the very first one create.  
All it does is set up the release document for a given release for the whole month, and then for there's ingestion.  
And then there's like ESRP scan, publish, approve and go live.  
And then sometimes there's an extra one or two, but these are mainly the different requests that we have code under.  
So ingestion is the first big one.  
Create isn't really anything special.  
It's literally the same for everything.  
You just make a release document, so for ingest this is where we prepare the data.  
So anything that we can be doing during ingestion in order to speed up the subsequent requests, so if published would be faster because of file is in a certain spot then we would put that file in that spot during ingestion.

 **Dina Helal** joined the meeting

 **Jacob Kissel** 2:41  
But we don't actually want anything to get to a customer endpoint during ingestion.

 **Mohit Gupta** joined the meeting

 **Jacob Kissel** 2:48  
So for example, if either an internal customer who's using an internally published piece of media, or an external customer with the public endpoint, we don't wanna be putting any data where that's actually accessible to anyone besides us, or at least getting pushed to anyone besides us before the other calls for publish and go live and things like that, we don't want to overload the operation.  
Additionally, it's because we have RM approval steps sometimes before publish and a lot for everything.  
We have arm approval steps before go live, so we don't want to accidentally push anything anywhere before the RM is actually approved.  
A given set of media.  
So ingestion across all of our update types mainly just looks like file copies and metadata preparations OA lot of our different update types have different pieces of metadata they require.  
For example, a lot of them require SHA 256 in order to.  
Publish to or ESRP scan or they require like file size for a PF we need the SHA 256 and the file size to feed it into the autopilot and pilot fish deployment system.  
O for in Jet for publish, we're actually publishing it there and we need those pieces of metadata during publish.  
But if we can get them during ingestion, then they're just already ready for the publish call.

 **Leslie Trowbridge** joined the meeting

 **Jacob Kissel** 4:25  
So if we can gather as much metadata about our media types before publish, it makes it so that when publish does get called later on, it will be faster and there will be less processing that has to do be done there.  
So ingestion is really just a big preparation step for the rest of our publishing and go live operations.  
So yeah, we don't want to actually push to any endpoints.  
Obviously that needs to be locked behind approve, especially ESRP scan.  
We don't want to push to any endpoints before ESRP scan is done.  
That's publicly accessible.  
So every time, for example, if we add a new update type and we need to write the ingestion step, we need to have a good understanding of what publishing looks like so that that can be done during publish and so that we don't accidentally do some publishing during ingestion.  
And sometimes that looks like a little bit of tribal knowledge.  
Sometimes it looks like you have to have kind of more knowledge about the update type itself.  
To understand why we're doing something in the ingestion step instead of the publish step.  
But these are kind of the overall tenants we use to talk about this request, this specific operation and understand it and kind of use a shared knowledge across the team.  
So even if I didn't know much about VHD, for example, if I were talking to Polo or Sam about the VHD ingestion step, I would understand I would be able to understand why they're doing something in the ingestion step versus why they're doing it in the publish step, for example.  
So just the two two examples of update types that have a little bit different things happening during ingestion for containers on the left, we actually upload the containers to do different places.  
One place is that we uploaded them from the Visual Studio drops to store to our storage account and from the storage account.

 **Leslie Trowbridge** left the meeting

 **Jacob Kissel** 6:23  
They get pushed into ESRP and then also from the storage account they get pushed into our private container registry.  
So for anyone that doesn't know much about containers, a container registry is basically how we publish containers and serve them to the public.  
And the only extra thing to know is that we have our own private container registry that's accessible by only us and only the public.  
Container registry team who eventually takes those images to the public.

 **Leslie Trowbridge** joined the meeting

 **Jacob Kissel** 6:52  
So some of the piece, some of the quote unquote folders or repositories in our registry are truly private to us and MCR doesn't even know about them.  
And the other one, MCR does know about and they'll take those from there and give them to the public.  
So during ingestion, we'll put them into our private folders in the registry and then later during go live, we'll put them into the public folders.  
But since they're already in the private folders, the go live step is a lot faster.  
O without getting into too much implementation details, this is kind of a a split where we're preparing the data to get copied because the copy is done during ingestion and then once it's actually executed during go live, it's a lot faster, but we do not put them into the public facing repositories of our registry until we're called to do go live.  
Umm, so containers has quite a few different pieces going on under the hood, so there's a couple extra file copies here, but for something like pass native right now our publishing responsibility is a lot more simple.  
So path for containers where this media types are going out to the public past native is actually just serving internal teams only.  
So past native our images that get deployed to be kind of the the underlying bones of Azure and other services inside of Microsoft.  
So PAS native doesn't go to any public consumers, they're all internal partner teams.  
O for past native, there's actually, uh, once we finish the internal publish step of past native, another team actually ends up taking the images and doing some extra stuff and deploying them to the actual agents that are going to be running those images.  
But for our current set of responsibilities, it basically is just a file copy for publish O the end goal of like where me pubs are responsibilities ends and the past native deployment team responsibility begins is once the files are on a specific agreed upon file share with a certain agreed upon naming convention. For those files the deployment team picks them up.  
So because that's the case for past native, we don't even do that file copy during ingestion, because that would be actually us being done with publishing as well.  
And right now, once ingestion is done, publishing starts immediately.  
So it's not really that big of a deal if we were to do it during publish, but eventually if we wanted to have an RM approval step in between ingestion and publish or we wanted to have a wait period or we wanted to only do it X days minus patch Tuesday, we would be in trouble.  
If we were doing that file copy during ingest instead of publish, so for past native actually all we do is aggregate the metadata for the different past native files that need to be copied.  
We figure out which file copy location they're going to be going to, but we don't actually do the copies, so we basically get all the metadata ready during ingestion for past native and then later during publish those file copies are actually executed.  
OK, these are kind of what I'm trying to illustrate here is just going from the implementation detail understanding of what the media types ingestion and publishing and go live looks like to see why someone.  
So for containers were doing like a file copy, but for past native we're literally just doing metadata in the file copy, stopping in publish.  
So we have more than just these two media types obviously have VHD, APF, ISO, ESD, umm and we even have a couple different endpoints that VHD's go to now.  
Now, so there's all all different types of operations happening during ingestion, but we're following the overall tenant of not publishing to a certain endpoint, but preparing as much as possible so that the publish to that endpoint is as fast as possible.  
So when we go to implement ingestion, since we're following these overall tenants kind of from a design perspective, we're also following a certain number of tenants from the actual code implementation perspective.  
O what this means is even though even though for past native we're only getting metadata, but for containers, we're actually doing some file copy operations.  
There's still a number of things that are shared interfaces between the two in our actual code, so we work with release documents, and for every request we get, there's a copy of a release document in our database.  
So we would have like a create document, one for everything and then every time we get an ingest we have an ingest release document O if I got ingest with a version 11 for containers I would have a release document for it and then if I got ingest for version 1.12 of like server 2022 containers, I would have another one and we get one release document per ingestion request.  
Basically, for every single update type after that, the release document has some metadata in it and all of our metadata.  
Now is coming from media seeker URLs, so our upstream team is HCC human centered computing I think is what that stands for and their system is called SMF, which stands for serviced Media factory and they're the team that's creating this media that we're taking and publishing to different endpoints.  
So smaps current mechanism for serving us, they're created metadata and actual files is through media seeker and media seeker is kind of an API that they expose to allow us to query to get the information from them.  
O when they create a set of a set of media, they'll send a sign off to AWE, and that includes the media seeker URL in it amongst a couple other things like the payload version of the media that they created and some stuff like that, and then AWE passes that media seeker URL through to us and then that is our main point of entry to get all of the metadata for whatever we need for our for our update types O we'll use that media seeker URL.

 **Deepak Kunwar** left the meeting

 **Jacob Kissel** 13:16  
We'll get all the metadata and we'll also do something with it for some of the stuff.  
So that metadata has like a drop name for containers and we'll use that drop name to download those container files that SMF created and do something with them or also media seeker URL contains metadata like the SHA 256 Digest or the file size or file names things like that.

 **Aravind Siddoju** left the meeting

 **Jacob Kissel** 13:41  
So the first couple steps of ingestion for most of our update types is always that we create release document and then we get the metadata.  
At least filter through it for the media, seek re URL and I'm going to show an example of this media seeker URL later.  
What its contents are?  
And then so these are the shared steps.  
And then once we get past the shared step of at least filtering down the media seeker data then excuse me.  
I'm sorry, after that it becomes a matter of actually doing something with it.  
And that's when each of the individual update type interfaces take over O containers.  
Will have an implementation and will have an implementation ABF Everything has its own different implementation based on what operations it needs to during ingestion.  
So they all have the same input, which is for all intents and purposes, this media seeker URLs data amongst a couple other things, and they'll do something different with that.  
And then lastly, we get to the last shared step, which is they have the same output which our release proposal metadata documents and this is very long winded.  
So we almost always just say RPMS, but basically for all of our ingestion operations, the last shared piece of it is that we create 1 RPM D per file that was ingested.

 **Mohit Gupta** left the meeting

 **Fabiola Ansara** left the meeting

 **Jacob Kissel** 15:04  
There are a couple I believe there may be one or two places where that specific paradigm is broken and an RCMP can point to a couple things, but for for almost everything we ingest one, we create 1 RPM D per file.  
So if I were to get a container ingestion request and I ingest 1 server 2022 containers churn, I would have one release document for that.  
Say it's like version 1.1 and then for server 2022 containers we actually publish 3 different types of them, nano, server, server and server core.  
So or version 1.1 of server 2022 containers we would actually have three different RPMS's created after ingestion is all said and done, one for a nano server, one for server and one for server core.  
And the RPMS's have a little bit more detailed metadata in them compared to the release document.  
So release document has very few fields but RPMS have a lot of fields and some of them could be specific to a different update type like for containers they have a field of that says where the containers metadata file is stored or sorry it's config and manifest files are stored which are pieces used later during ublish thing.  
For for a PF and past native, there's a field that says where they're actually gonna end up getting published to.  
Like what?  
The folder location is, so these are PMD's are basically where we store the outputs of all of that gathered metadata during ingestion, and then later on during publish, go live and everything.  
The RPMS are actually what we use to do those operations.  
The release document lets us kind of tie these together in correspond to them, but the RPM's are what actually has all the important information in them that's used during publishing operations and go live and stuff like that.  
So that's kind of the last of the initial overview of the ingestion and shared and differing in ingestion interfaces and stuff.  
So I was gonna show an example of sifting through a media seeker URL and then if we have time I was going to show a couple of these interfaces in code.  
But before we get to that, does anyone have any questions about the overall idea of the ingestion shared and differing stuff?  
Go ahead, achal.

 **Achal Shah** 17:31  
OK.  
Yeah.  
So you talked about like there's a lot of common things that happen, right for all media types, but there's obviously differences as well.

 **Jacob Kissel** 17:39  
Umm.

 **Achal Shah** 17:44  
Now as far as the ingestion services goes, is is it 1 service to channels all the media or do we break it up into different services for different media types?

 **Jacob Kissel** 17:44  
Yeah.  
Yes.  
So so as it stands right now, there's one ingestion service because there's a good amount of stuff that's shared, it makes sense to put all of that in one ingestion service and then inside of that service, there's one handler per update type.  
So there's one shared ingestion handler which is the entry point, or I should say controller that's the entry point for an ingestion request and it does a certain amount of processing that applies to every single piece of our update types.  
And then after that it goes and figures out the handler it needs based on the update type, and that handler still lives on the same ingestion service.  
Later on, during an publishing, every every update type has its own publishing service because it's a lot more specific.  
Umm to what it needs to be doing, but the as it stands right now, all of the ingestion workflows are on one ingestion service and all of the rest of the like publishing and approve and go live and stuff.  
Those are all on their own services, umm, but it would totally be possible to move that shared ingestion code into a library and move each of the ingestions into their own services as well.  
I think that would be a totally valid design as well for this setup.

 **Achal Shah** 19:14  
OK.  
And and one other question with ESRP scans, is that considered part of ingestion or publishing or something in between?

 **Jacob Kissel** 19:21  
Oh, yeah.  
So yeah, yeah, that's a good question.  
I didn't get exactly to that, so basically.  
ESRP is kicked off at the very end of ingestion, but ESRP lives in kind of a limbo space because, uh, it's kicked off as its own operation.  
In fact, we even have our own ESRP scan document, but we kick it off ourselves after ingestion is done, and if ingestion succeeds, it actually doesn't imply the success of ESRP scan, because ESRP scan can sometimes be a very long running operation, it can be 3 hours or it can be 24 hours, depending on the file size, depending on their queues and everything like that.  
So basically we kick off ESRP scan at the end of ingestion and then we have a different point in time in which we pick U the results of the SRP scan.  
And that is for most of the update types.  
That's room approval.  
So if we get to the point in time one or a couple days or three days or four days before Patch Tuesday and the RM approves a certain version of a media that we have ingested already, if we go into our documents and ESRP has not completed, that is the main signifier to fail RM approval.  
So I don't know if I can't remember if the room gets a message itself, it almost never happens because ESRP is almost always done by the time armor approval happens.  
But basically, the approval operation itself will fail and at the very minimum and ICM will be raised saying, hey, we try to approve of version of something whose ESRP scan has not finished.  
So it's kind of an asynchronous lineup there where we kick it off during ingestion, but we let it run in the background while we do other stuff and then eventually we go and check the results once the ARM approval triggers that.  
Any other questions before I look briefly at the the example of the media seeker URL?  
Umm, I will go ahead and going to open a notepad instance really quickly.

 **Achal Shah** 21:50  
If you're sharing something, we can't see it yet.

 **Jacob Kissel** 21:55  
Uh, yeah, I stopped sharing on accident, but I'm I got the thing open, so I'm gonna go start again one more time.

 **Achal Shah** 21:56  
OK, OK.

 **Jacob Kissel** 22:00  
Sorry.

 **Achal Shah** 22:00  
OK.

 **Jacob Kissel** 22:02  
OK.  
Yeah, so.  
Basically, when we get a request or when we get the ingestion request from AWE, it has this URL.  
Uh.  
Here it doesn't have this type equals container thing, but that's something we do on our side to speed up the the query sometimes.  
But basically this is this is the section of the URL that it will have and we'll go in here and this has very high level information and metadata about the drops.  
And another thing to know about the media seeker URLs is there's a contract that says technically SMF can put whatever they want.

 **Leslie Trowbridge** left the meeting

 **Jacob Kissel** 22:46  
Uh, whatever number of update types and medias they want in this, as long as when we use our filters we find the ones we're looking for.  
So like this is for RS1 containers.  
Basically we have a list of drops that we're expecting and we expect to find all of those in this list, and there should be no more than one of each one.  
So like we know that one of the drops we're looking for is this string plus E which is like the language at the end.  
Umm, if there's more than one of these, then it's a problem because we don't know which one to choose, and if it's not existent then it's a problem because it should be there.  
But for example, if this one actually only has containers in it cause of the filter at the end.  
But sometimes there can be extra drops in there that either SMF created as like just a random output of their build, or maybe they batched builds together.  
Batching is an important thing that they do, so we do have to have some filtering logic set up for this.  
So basically, we'll go through this list of updates or these list of medias and then we'll get it to drop ID and each one of these is a different drop ID.  
They all share the same build GUID and then we have to do one more query on media seeker.  
Umm, which is just putting the drop ID here and I believe this is how you do it and then this will give you a little bit more detailed information so you can see on this side we're looking at server data center core PT.  
I think this is Portuguese, but I don't know for sure.  
And then the drop ID.  
We use this drop ID to query the exact drops information and this has some extra stuff in it that wasn't in the other page.  
This is for usage purposes, making it more efficient to call the other URL because if they have to put more information in that one it will be slower and this information actually isn't just stored directly as like a queryable, queryable table, it's actually calculated on the family from different places in their system oonce we query this, they kind of go and get all this information and then the important stuff from here mainly is this cloud drop name.  
And this is actually the location of the file that we need to ingest.  
So this can be put into drop.exe.  
Uh and or use.  
Right now we use ADO pipeline to ingest this, but basically this drop name is used if we were to go get this drop name and anddrop.exe we would see the actual container files in there and we would be able to download them as well as the other container metadata and then some of the other important stuff like size and megabytes I think is used for some of the update types we used to care about this flavor because containers used to arm 32 as well, but now everything is AMD 64 But I think this is actually relevant for VHD's is one of the filter pieces.  
Umm, excuse me, but yeah.  
So the main important thing here is querying down to find the cloud drop name for each of the updates.  
So like for server core, actually every single one of these is one that we're ingesting for 1607 because it has like 17 different layers that we ingest for, like for server 2022, there's literally only three, but this is kind of the overview of the media seeker URL set up.  
It's almost the same for everything else, where we're all we really care about getting out of this is the cloud drop name, but sometimes there's a couple extra pieces of metadata.  
I don't think I have much time to go into the code.  
All I really wanted to show was like the actual media ingestion.  
Service.  
So if I were to open.  
Yeah, this is actually it right here.  
So I don't know if you can not sure how to zoom in on this, but basically we have the service here and then media data handler.  
All of these are different update types, so Autopilot data handler, ESD, ISO.  
And then panned and then server container and then there's a generic one here.  
So this generic one actually has just to briefly show it.  
This has the the actual media seeker filtering logic, so like this function retrieve file Uri path umm a lot of these are used by all of them and then let's see if I can find it. Yeah.  
So filter metadata is one that's shared.  
So this one will take in the release document and this media archive items is actually the list of all of the items that we see in the media seeker URL on the original call.  
And then so the generic media data handler has logic that can filter and just find the ones for our specific flavor and media type name and things like that.  
So there's quite a few shared pieces of logic in this one, and then once you look at each individual data handler, say something is failing during container ingestion and it's not one of the shared things.  
The first place that we would look is in the actual data handler for containers and all of these are much bigger, so this has all of the container specific ingestion logic and then same for like past native.  
It's a lot smaller because it's literally just doing metadata, but that's without going too much into the code.  
And also we're out of time.  
That's the initial that's just showing what the current ingestion setup is, where there's some shared stuff, and then it goes in and has each of its update types owned data handlers.  
Anybody else have any questions before we close?

 **Achal Shah** 28:41  
The only other thing I would add is that this is kicked off by a logic app, correct?  
And then once the yeah, once ingestion is done, I guess a call back goes back to the logic app and then the logic app does whatever else it needs to after you know.

 **Jacob Kissel** 28:48  
Oh yeah.  
Mm-hmm.  
Yeah.  
Yeah, exactly.  
So almost all of our operations are kicked off initially from a logic app because logic apps are allowed to wait a long time and some of our ingestion operations can take 3 hours.

 **Achal Shah** 29:07  
Yeah.

 **Jacob Kissel** 29:09  
So like this controller will take in a callback URL from the logic app.  
It will do this task factory start new thing, it'll save the inner callback URL.  
So it's there on the thread.  
Umm.  
After it's done, it will go to the end.  
UM, if an exception is caught, it'll send something to the logic app otherwise oh, this return accepted thing.  
By the way, this happens right when the task factory started.  
It returns accepted to the callback URL and then eventually the callback URL like for example.  
Here we post a response to that callback URL to the logic app because it can wait it's set up as a web hook.  
So if it was just a normal HTTP request, it would time out after like 2 minutes, but instead we send it an accepted immediately and then we send it a call back URL.  
Maybe even 24 hours later, if it takes that long, it shouldn't really.  
But umm, that's?  
Yeah, that's kind of the entry point for the ingestion.

 **Achal Shah** 30:17  
Great.  
Thank you, Jacob.  
I know was very, very short notice.  
I asked him yesterday and he put all this together and yeah, I guess anybody, if you have any further questions on this, you know, feel free to reach out to Jacob over anybody in the media team, yeah.

 **Jacob Kissel** 30:21  
Yeah.  
No worries.  
Yeah.  
Yeah, please do.  
All right.  
I appreciate everyone's time.

 **Achal Shah** 30:41  
Hey, thank you.

 **Dina Helal** 30:43  
Thanks not hey back.

 **Kush Mishra** 30:44  
And you can.

 **Jacob Kissel** 30:46  
Thanks everyone.

 **Lacastan Moodley** 30:46  
Thanks.

 **Jacob Kissel** 30:46  
Bye bye.

 **Mustafa Mizrak** 30:47  
Thank you.

 **Dina Helal** left the meeting

 **Lacastan Moodley** left the meeting

 **Dominik Guzowski** left the meeting

 **Kush Mishra** left the meeting

 **Tavishi Gupta** left the meeting

 **Pinar Sen** left the meeting

 **Jacob Kissel** left the meeting

 **Mustafa Mizrak** left the meeting

 **Bhaskar Verma** left the meeting

 **Achal Shah** stopped transcription