Topcoder Project Input Form – CIM Outbound Mail Processor

App Summary

This is an API/process which will watch a shared network directory for files it needs to process as part of a workflow; the current known workflow steps are to place item into our long-term storage API (DMS), fire a "Item Mailed" event, clean up the artifact on the filesystem

The functional description is simple but a relative high degree of fault tolerance and traceability of work/tasks done is desired.

Throughout this document the system will be abbreviated as COMP.

Defined Terms

This section lists some nouns/terms which will be used throughout this document and related documents.

Term	**Definition**
COMP	CIM Outbound Mail Processor, the SUD
Mailed Item Bundle [MIB]	Our OMS software will, upon successful processing, output to the watched directory two files identically named with different extensions: the artifact (PDF file) and a CSV file which provides metadata about the item which was mailed.
Mail Processing Job [MPJ]	A Mail Processing Job is the top-level unit of work that is to be done. Against this job we will track overall success/failure of the internal workflow steps/tasks.
Mail Sent Queue [MSQ]	The place in which OMS stores the MIBs. The CIM Intake Channel will place the artifact into the MSQ and create a pending Job; the metadata is coming from CIM in this case, not OMS OCR process. **NOTE:** The first *real* implementation will be a shared network directory but the software solution should abstract this to the point where we could at a later date change design and put the MIBs into another structure (e.g. AWS queue). The shared network directory requirement is being driven by the COTS mail processing software OMS.
OMS	Our COTS mail processing software
DMS	Our internal API providing a surface into long-term storage; clients provide the item to be stored and metadata about the item for later retrieval.
Enterprise Event API	This is an internal API to which we POST "events". In this case, will be

	posting a Postal Mail Sent Event.
COMPDB	It’s recommended that our design selects MIBs from the directory but merely places it into COMPDB to support asynchronous processing, retry, error handling, *etc*.
Process Mail Workflow Task	A conceptual name for a definition of “something to do in relation to this MIB”; e.g. upload to DMS, fire postal mail sent event, clean up source MIB. The tasks will need to be processed serially; upload to DMS first, fire postal mail event second. The system will keep track of every time we attempt to complete a Task; for each Task Attempt we will track the time we attempted to execute the Task and will capture Error details on failure.
${CONF_}	This is a convention which indicates a configurable value is being referred to.
${}	Calculated value, logical name.
Communications Interaction Management (CIM)	A family of internal systems which manage all communication touchpoints with our customers.

Workflow

Please describe the ideal workflow for the proposed solution.

Generalized summary:

- New items arrive in MSQ
- Create MPJ in COMPDB from MIB information (OMS OCR Intake Channel) or correlate the new item to a Pending Job (CIM Intake Channel)
  - Delete MIB from MSQ
- Process will have internal “job processors”, which watch for MPJs which need processing
  - Serial Tasks:
    - 1 – Upload to DMS
    - 2 – Fire Postal Mail Sent Event
      - Calculate overall completion status/success status for the just-processed MPJ
        - On fully completing a job, the artifact is removed from the COMPDB to avoid accumulating raw contents of all processed mail items; the remaining info is kept for historical/auditing purposes

MIB Intake Workflow Step #	**Description**
Step 1	COMP notices an MIB in MSQ; that is, one which is not in our COMPDB job storage
Step 2	COMP stores MIB in COMPDB. Initial state: CompletionStatus=INCOMPLETE, SuccessStatus=NOT_ATTEMPTED, 2 internal Tasks (types: Upload to DMS, Fire Postal Mail Sent Event) with NOT_ATTEMPTED statuses for each, Remove MIB from MSQ

MIB Processing Workflow Step #	**Description**
Step 1	COMP selects MPJs which need to be processed Criteria: CompletionStatus != Complete

	LastTaskExecutionTimestamp > ${CONF_TASK_PROCESSING_SPAN_MINUTES}
Step 2	COMP executes any remaining outstanding workflow tasks (UPLOAD_TO_DMS, FIRE_EVENT)
Step 3	ON SUCCESS OF ALL WORKFLOW TASKS Set CompletionStatus to COMPLETE Set SuccessStatus to SUCCESS Set LastProcessingTime to ${now} Increment ProcessingAttempts Set TTL for MIB to ${today+${CONF_TTL_JOB_SUCCESS}} ON FAILURE If ProcessingAttempts !> ${CONF_PROCESSING_ATTEMPTS_MAX} Set CompletionStatus to INCOMPLETE Set SuccessStatus to FAILURE Increment ProcessingAttempts Set LastProcessingTime to ${now} If ProcessingAttempts > ${CONF_PROCESSING_ATTEMPTS_MAX} Set CompletionStatus to COMPLETE Set SuccessStatus to FAILURE Increment ProcessingAttempts Set LastProcessingTime to ${now} Set TTL for MIB to ${today+${CONF_TTL_JOB_FAILURE}}

Functional Attributes, Expectations, Capabilities

This section lists some functional attributes, expectations and capabilities which may not be apparent from prior sections.

Functional Attribute/Expectation/Capability	**Description/Discussion**
Abstractions	The system should abstract to allow implementation change for at least the following considerations: • Mail Sent Queue

	• COMP Work State Storage
Internal Tasks	Each item we capture as outbound mail must have a series of steps performed, against which we will need to support retry semantics. We have two immediate concrete Tasks defined, but the implementation should allow us to define additional types in the future (in code, not dynamic).
Job Tracking/Querying	While this API does not feature a UI, developers and runtime support should be able to perform queries as to the status of a certain MIB processing state, MIBs which ended with a final FAILURE state, etc.
Internal Timing	The app/API should be self-contained; it will not be getting signaled to sweep for work to perform, nor will it have access to cron shell utilities.
Cluster-Safe	While the initial deployment scenario will feature a single Node process, care should be afforded in safely picking up and processing work. It is noted that, in relation to working against a shared network directory, this can be tricky.
Avoid Duplicates	The system should guard against trying to process the same MIB > once
API Capabilities	The API surface should provide: - MPJs within date range - MPJs within date range with selection of current completion status (i.e. complete, incomplete) and selection of overall status (i.e. success/failure) - Create MPJ

Integration Points

Will this application be dependent on data from another system or tool? If yes, please briefly describe that dependency here. This can include integration with an API or an existing backend/database.

Integration Point	**Description**	**Required**
API – DMS	DMS is our current API which provides long-term storage of objects. A mock request/response handler will be provided.	X
API – Enterprise Events	This internal API is how we 'fire' internal business events. A mock request/response handler will be provided.	X
Work Queue	Current target to scrape for MIBs is a shared Windows NTFS directory. External code will not have access to Wellmark-internal network share.	X

Technology Stack

Item	Description
Programming Languages	Typescript
Frameworks	NestJS
Database	MongoDB • Database Name: outboundMailProcessorDb • Collections: ○ jobs – Storage for MPJ objects
Server	Node 12
Hosting Environment	Docker container

Configuration Variables

The following table lists the information regarding required configurable switches, their description, datatypes, and defaultable status. It is expected that we will be able to configure or override these values using standard environment variables.

NOTE: All variable names should be prefixed with "WM_OMP_" ("Wellmark Outbound Mail Processor"); this enables us to visually "group" all vars used by this process when enumerating/querying the environment.

Name	Purpose	Type	Required	Default Value
PROCESSING_ATTEMPTS_MAX	Provides an upper limit as to how many times the job processor will attempt to complete each task.	Integer	Yes	10
TASK_PROCESSING_SPAN_MINUTES	Specifies how many minutes need to elapse from "last processing time" where the job	Integer	Yes	10

	processor will consider executing the tasks in a job.			
SYSTEM_NOTIFICATION_MAIL_ADDRS	We don't have rich capabilities for system monitoring + notification. This is to handle notifying interested people on certain error conditions (e.g. failure to delete MIB)	String[] Format: Email Addresses	No	
PENDING_TTL_MINUTES	The number of minutes a MPJ can be in a "pending" state before being marked as orphaned.	Integer	Yes	2160
DATASTORE_TYPE	Indicates type of datastore	String	Yes	MONGODB
DATASTORE_HOST	Hostname of datastore	String	Yes	
DATASTORE_PORT	Port number	Integer	Yes	For MONGODB, 27017
DATASTORE_DB_NAME	The name of the DB	String	Yes	outboundMailProcessorDb
DATASTORE_USER	Username for DB	String	Yes	
DATASTORE_PASSWORD	Password for DB	String	Yes	
DATACENTER_ENV	Which datacenter this app is operating in; defined in Node dependency `wm-lib-runtime-enviroment`	String	Yes	ONPREM
NODE_ENV	The staging	String	Yes	

	environment; defined in dependency `wm-lib-runtime-enviroment; values are LOCAL, DEV, SIT, UAT, PREP, PROD`			
ROOT_URI_DMS	The root URI for contacting the DMS API	String	Yes	
ROOT_URI_EVENT_API	The root URI for contacting the Event API	String	Yes	
ROOT_URI_CIM_COMM_API	The root URI for contacting the CIM Communications API	String	Yes	

Error Conditions

This section lists some known error/failure conditions which the code or system will need to be concerned about. The default action on all errors/conditions in-code will of course use the app log's ERROR stream to capture details about the failure in addition to any handling steps provided below.

We will also want to consider (in our job processing routines) whether an error is retryable or not; this will directly impact how we calculate which statuses to apply to the current job (completion status, success status).

For example, if we fail on uploading to DMS due to a timeout error, that would be retryable since it'd be expected the service will be contactable in the future; however, if the DMS API returns an error which indicates we're trying to do or store something illegal (returns a 400 Bad Request), it's likely the code/job is at fault and we're not going to succeed no matter how many times we retry the request.

Condition	**Description**	**Handling**
Can't upload to DMS		Consider retryable; calculate statuses as appropriate. Will not continue any tasks "after" DMS upload.
Can't create Event	This is where the system will raise an event notifying other systems that we've sent some postal mail.	Calculate statuses as appropriate
Orphaned MIB Artifact	OMS has written an artifact file but it didn't write a corresponding metadata file; we should build a reasonable but short tolerance to consider something "orphaned"; 2 minutes should be more than enough on intake (OMS should be writing in tandem on processing).	Mark Job as Complete/Failure status, capture cause.
Orphaned MIB Metadata	OMS has written a metadata file with no corresponding artifact.	Mark Job as Complete/Failure status, capture cause.
Invalid Metadata	OMS has written a metadata file which we can't parse or missing required data.	Mark Job as Complete/Failure status, capture cause.
Orphaned Pending Job	The CIM Intake Channel will write the item to OMS' intake directory but directly call this API to create a new MPJ with all the metadata needed for further processing. This could be called a Pending Job. If the artifact being mailed does not arrive within a given period (${CONF_PENDING_TTL_HOURS}), the Job orphaned.	Mark Job as Complete/Failure status, capture cause.

Cannot Store/Create MPJ	Save to persistence layer failed	Log to error stream. CHALLENGE: If we leave the artifacts in place, we will need to notice that; mailroom wants us to be clearing these directories as we process.
Cannot Remove MIB		Notify ${CONF_SYSTEM_NOTIFICATION_MAIL_ADDRS} if not empty, else just log to error stream.
No Metadata Captured	If OMS is OCR-ing the metadata, there is a chance that it could not successfully scan any fields as it's a positional determination.	The job will be marked as COMPLETE/FAILURE and we'll capture the reason as "missing required metadata". The object in the DB will point out the reason as well as the filename(s) so we can have a running record of the letters failing OCR. The MIB will be deleted on MPJ creation.

* Not an exhaustive list.

Model Objects

This section provides some *possible* examples of how we can structure the Jobs and what the IO models are for integrations. What that means is that the model dealing with Job handling/retry/*etc.* may mutate as development begins while the surrounding items (request context, relationships) would need to stay; you'll notice the data movement into and out of the system.

Model: MailProcessingJob

The intent of this model is to capture all the data *about* a Job (the content, the addressee, the relationships, *etc.*) with the Tasks needed to *complete* a Job (*e.g.* uploading into long-term storage).

This is a single-object view of a single complete Job. From this one should be able to extrapolate what a "failed" job could be represented.

```
{
  "version": "1",
  "requestContext": {
    "source": {
      "system": {
        "name": "some-app-or-api",
        "version": "1.0",
        "stagingEnvironment": "DEV",
        "datacenterEnvironment": "AWS"
      },
      "user": {
        "id": "bob1234"
      }
    }
  },
  "id": "310d58ad-aa25-409e-9135-12a7f48b95d4",
  "created": "2020-02-11T15:05:07+0000",
  "name": "CIM_DEV_B-9619649_IOI4444444-1581447907",
  "completionStatus": "COMPLETE",
  "overallStatus": "SUCCESS",
  "lastProcessingAttempt": "2020-02-11T19:15:07+0000",
  "requestedSendDate": "2020-02-11",
  "recipient": {
    "addressee": "JON GRUDEN",
    "addressLine1": "3333 Al Davis Way",
    "addressLine2": "",
    "city": "Las Vegas",
    "state": "NV",
    "zip": "89118",
    "zip4": ""
  },
  "item": {
    "contentType": "application/pdf",
    "encodingType": "base64",
    "content": "base64_encoded_bytes_of_the_pdf_here"
  },
  "taskOrder": [
    "UPLOAD_TO_DMS",
    "FIRE_EVENT_POSTAL_MAIL_SENT"
  ],
  "tasks": {
    "UPLOAD_TO_DMS": {
      "completionStatus": "COMPLETE",
      "overallStatus": "SUCCESS",
      "lastProcessingAttempt": "2020-02-11T18:58:00+0000",
      "attempts": [
        {
          "time": "2020-02-11T18:48:00+0000",
          "status": "ERROR",
          "retryableStatus": true,
          "errorDetails": {
            "name": "DMS_RUNTIME_ERROR",
            "message": "DMS not responding",
            "details": "here we would put the stack trace (need to make it serializable if
caught exception), service response, etc."
          }
        },
        {
          "time": "2020-02-11T18:58:00+0000",
          "status": "SUCCESS",
          "successDetails": {
            "details": "here we would put DMS service response (if relatively
small/serializable); something to keep track of to capture what DMS responded with."
          }
        }
      ]
    },
    "FIRE_EVENT_POSTAL_MAIL_SENT": {
      "completionStatus": "COMPLETE",
      "overallStatus": "SUCCESS",
      "lastProcessingAttempt": "2020-02-11T19:05:00+0000",
      "attempts": [
        {
          "time": "2020-02-11T19:05:00+0000",
          "status": "SUCCESS",
          "successDetails": {
            "details": "here we would put event service response (if relatively
small/serializable); something to keep track of to capture what DMS responded with."
          }
        }
      ]
    }
  },
  "storage": {
    "system": "DMS",
    "conf": {
      "locationId": "555",
      "locationName": "some_group_name"
    }
  },
  "relationships": [
    {
      "type": "CRM-CASE",
      "conf": {
        "id": "020e2e43-cb09-4849-9ee0-c8c13b12d0bb"
      }
    },
    {
      "type": "CONTENT-SOURCE",
      "conf": {
        "sourceSystemType": "GMC",
        "catalogId": "gmctid:610d58ad-aa25-409e-9135-12a7f48b95b3",
            "formId": "B-9619649"
          }
        }
      ]
    }
```

Web Service Endpoints

While most of the code for this API process will be internal job processing/coordination, we will need the some API endpoints.

Create New Job - Input

In the case of MSQ Input Channel of CIM, the Postal Mail API will 1) write the file to a folder OMS is monitoring and 2) call COMP to create a new job in “pending” status.

Endpoint: POST /job
Payload:

```
{
  "version": "1",
  "requestContext": {
    "source": {
      "system": {
        "name": "some-app-or-api",
        "version": "1.0",
        "stagingEnvironment": "DEV",
        "datacenterEnvironment": "AWS"
      },
      "user": {
        "id": "bob1234"
      }
    }
  },
  "name": "CIM_DEV_B-9619649_IOI4444444_1581447907",
  "recipient": {
    "addressee": "JON GRUDEN",
    "addressLine1": "3333 Al Davis Way",
    "addressLine2": "",
    "city": "Las Vegas",
    "state": "NV",
    "zip": "89118",
    "zip4": ""
  },
  "item": {
    "contentType": "application/pdf"
  },
  "storage": {
    "system": "DMS",
    "conf": {
      "locationId": "555",
      "locationName": "some_group_name"
    }
  },
  "relationships": [
    {
      "type": "CRM-CASE",
      "conf": {
        "id": "020e2e43-cb09-4849-9ee0-c8c13b12d0bb"
      }
    },
    {
      "type": "CONTENT-SOURCE",
      "conf": {
        "sourceSystemType": "GMC",
        "catalogId": "gmctid:610d58ad-aa25-409e-9135-12a7f48b95b3",
        "formId": "B-9619649"
      }
    }
  ]
}
```

Some notes:

- Notice that many of the properties of the ContentJob object model is here provided by the external caller
    - Request context of caller
    - The name of the file which we're writing for OMS to process
        - The name is "namespaced" as: CIM_DEV_${formId}_${letterSubjectIdentifier}_${unix_epoch_time}
    - The letter recipient
    - The content type of the item we'll be picking up when moving from "pending job" to "in-process job"
    - Where we'll be storing the item
    - Relationships – This will be used by downstream systems, COMP will merely be echoing these out into our event system
- The "relationships" is a simple dictionary/map structure which may be empty; it is the responsibility of the clients of COMP to indicate their relationships (which are, again, used by systems downstream); for example, if we're mailing something *not* in relation to a CRM case, there wouldn't be a "CRM-CASE" relationship provided in the example

Create New Job – Outputs

Success

HTTP 200
COMP will have generated a guid as the top-level job ID; in the example above, the system would be returning `310d58ad-aa25-409e-9135-12a7f48b95d4`

MSQ Input

Our mailroom software (OMS) will be outputting MIBs into a shared network directory structure and is accessed via UNC path in the software. We will not be able to externalize this directory; upon acceptance, our plan is to run the software with the requisite account permissions to achieve RW into the specified root directory.

Depending on the Intake Channel, we will process the MSQ in differing ways:

- OMS OCR
  - OMS OCR writes MIB to MSQ
  - COMP accepts MIB to create MPJ
- CIM
  - CIM will call COMP to create a “pending job”; the pending job will be correlatable by filename and jobs coming in via this process will have a consistent name prefix
    - Upon the file landing in MSQ (from OMS), COMP will look up the corresponding pending MPJ and continue processing

OMS OCR Intake Channel - CSV File

The following table lists the CSV, in order. The CSV files will be written with one item per file accompanied by column titles.

Ordinal	Column Name
1	Customer ID
2	AddressBlock.Line1
3	AddressBlock.Line2
4	AddressBlock.Line3
5	AddressBook.Line4
6	AddressBook.Line5
7	AddressBook.Line6
8	FormID.Left
9	FormID.Right

Processing the CSV File

The OMS OCR process is imprecise and somewhat unpredictable due to the company not having one consistent standard regarding where we place our address blocks, whether we include a date, *etc*. Until this gets cleaned up, the OCR process will be producing some output we’ll need to try to handle systematically.

The goal of the OMS OCR process is to extract *at least* the address information from a given letter; if we do not get a “good” address, the MPJ should be marked appropriately as an error and cause captured.

The facts:

- Some letters will include a date in their first position (e.g. “February 5, 2020”)

- Some will not; these would start with Addressee Name
    - To be followed by Address Lines
    - City/State/Zip is combined into one line and surrounded by quotation marks (e.g. “Nevada, IA 50201”).

An example:

```
Customer ID,AdressBlock.Line 1,AdressBlock.Line 2,AdressBlock.Line 3,AdressBlock.Line
4,AdressBlock.Line 5,AdressBlock.Line 6,FormID.Left,FormID.Right
,"February 5, 2020",JON GRUDEN,3333 Al Davis Way,APT TD,"LAS VEGAS, NV 89118",,,707-
FM-1D
```

In this case, we have received the date before addressee name.

The output from this effort will be getting emitted into the Postal Mail Sent Event, the payload for which will be provided further down this document. From *only this section*, however, the desired datamodel would be similar to:

```
{
 “date”: “February 5, 2020”,
 “addressee”: “JON GRUDEN”,
 “addressLine1”:  “3333 Al Davis Way”,
 “addressLine2”:  “APT TD”,
 “city”: “Las Vegas”,
 “state”: “NV”,
 “zip”: “89118”,
 “zip4”: “”,
 “formId”: “707-FM-1D”
}
```

Observations:

- The date sometimes being in the first position vs. addressee name may be “easy” to handle since if there are numbers present, we could assume date
- The address lines will be much trickier; open to suggestions as to how to calculate confidence in good address capture
    - For example, if there were no “APT TD” value, the city/state/zip concatenation will take the place
    - We could perhaps find the “last” address line by recognizing ZIP pattern, recognize the “first” address line logically by “first after addressee name”, then filling in the middle?
- The FormID will either be in left, right, or neither; we will end up with a nullable singular FormID

CIM Intake Channel – POST Input, Job Creation

When accepting items from the CIM channel, the CIM process will call COMP to create a pending MPJ with the expectation OMS will be writing a file to MSQ within a specified timeframe

Integration with DMS

DMS is our API which provides long-term storage capabilities; in the case of API usage, we will be POSTing the base64-encoded serialized bytes of the mailed item along with accompanying metadata for later processing/retrieval from DMS.
s

Naming

We will be providing our own document key to DMS; this comes from the MailProcessingJob's "name" property.

Saving a Document to DMS - Input

Operation: `POST /documents/upload`
Input Model:

```
[
  {
    "file": {
      "bytes": "string",
      "url": "string",
      "bucket": {
        "bucketName": "string",
        "bucketKey": "string"
      },
      "contentType": "string"
    },
    "metadatas": [
      {
        "locationId": 0,
        "locationName": "string",
        "templateId": 0,
        "fieldArray": [
          {
            "id": 0,
            "fieldName": "string",
            "fieldValue": "string"
          }
        ]
      }
    ]
  }
]
```

In our use case:

- COMP will only be POSTing a single document per call
- We will be supplying the bytes, not "url" or "bucket"
- The single metadata object we will be supplying will mostly be the MaiProcessingJob object model *without* the Task processing details
    - Flattened to conform with the structure above

If we look back at the Job model, we'll find a "storage" property; if it's DMS (our one use case right now), "conf" will contain "locationId" and "locationName":

```
  "storage": {
    "system": "DMS",
    "conf": {
      "locationId": "555",
      "locationName": "some_group_name"
    }
  },
```

Notice the "metadatas" item; at the top level, we will be taking the properties from "storage.conf" and putting them into the place defined by DMS.

The following table lists the field array we'll be passing to DMS. The field values come from the model examples in this document.

Name	**Value**
mailProcessingJob.id	310d58ad-aa25-409e-9135-12a7f48b95d4
mailProcessingJob.created	2020-02-11T15:05:07+0000
recipient.addressee	JON GRUDEN
mailProcessingJob.name	CIM_DEV_B-9619649_IOI4444444_1581447907
crm.caseId	020e2e43-cb09-4849-9ee0-c8c13b12d0bb
content.source.id	gmctid:610d58ad-aa25-409e-9135-12a7f48b95b3
content.source.formId	B-9619649
uploader.system.name	Outbound Mail Processor
uploader.system.version	${version_from_codebase_package}
uploader.system.stagingEnvironment	${CONF_STAGING_ENVIRONMENT}
uploader.system.datacenterEnvironment	${CONF_DATACENTER_ENVIRONMENT}

Integration with Event API

Upon successful save into DMS, we need to emit an event to the enterprise systems which indicates we've sent a postal mail item and it's now in long-term storage. We'll do this by POSTing to the Events API.

Operation: POST /event
Input Model:

```
{
  "effectiveDatetime": "",
  "classification": "TECHNICAL",
  "description": "An item has been sent through postal mail",
  "subjectAreaNm": "event",
  "subjectSubAreaNm": "contactevent",
  "objNm": "CONTACT_POSTAL_MAIL",
  "srcSysCd": "????",
  "eventPayload": {
    "contentType": "application/json",
    "contentEncoding": "7bit",
    "name": "CIM_DEV_B-9619649_IOI4444444_1581447907",
    "extension": "json",
    "payloadObject": {
      "version": "1",
      "mailProcessingJob.id": "310d58ad-aa25-409e-9135-12a7f48b95d4",
      "mailProcessingJob.created": "2020-02-11T15:05:07+0000",
      "addressee": {
        "name": "JON GRUDEN",
        "address": {
          "line1": "3333 Al Davis Way",
          "line2": "APT TD",
          "city": "Las Vegas",
          "state": "NV",
          "zip": "89118",
          "zip4": ""
        },
        "wid": "UNKNOWN",
        "ioi": "UNKNOWN"
      },
      "item": {
        "link": "//some//link//to//dms//doc"
      },
      "mailProcessingJob": {
        "description": "put the full mail processing job here, sans
item & task tracking info"
      },
      "originalRequestContext": {
        "description": "if it comes through the CIM channel, we'll be
passing over a JSON object as original request context; put that here"
      }
    }
  }
}
```

NOTE: The yellow highlighting indicates where you can put verbatim the values presented; the rest will be tied to the job.

Specifications / Code

A section which lists the IO object models, API service mocks, and code which Wellmark will need to provide before development can begin.

User/Role	**Description**
Work Queue Input Definitions	This is the metadata portion of the MIB and will be a CSV (non-CIM intake path) or JSON (CIM intake path).
Postal Mail Sent Event – Object Model	Define the POSTed object which will go into the Wellmark Event API.
API Mock – DMS	Mock the ‘save item’ service endpoint in DMS; provide success and error responses
API Mock – Enterprise Events	Mock the Event API’s ‘fire event’ endpoint; provide success and error responses.
API Starter Code	The standard API baseline/starter for Wellmark APIs.
API Mock – CIM – Send Internal Email	Mock the endpoint which allows us to create internal mail messages