|  |
| --- |
| **Step-0**  **Covert Word Document To Html**   1. Combine tables and images to a single image 2. Convert to HTML and MHTM 3. Rename the html document 4. Create context json 5. Package all the files including word document into a zip file. |
| **WordToHtmlConvertor (conversion\code\wordToHtmlConvertor\wordToHtmlConvertor.py)** |
| Unzip ePI document ZIP file to work directory. |

|  |
| --- |
| **Step-1**  **Create Working Directory**   1. Create folder structure in the work directory based on the report domain, Procedure Type, medicine name , language and zip file timestamp 2. All the above details are extracted from the zip file name. 3. This end folder of the timestamp acts as the root folder for publishing all the results and intermediate files while converting this zip file html. |
| Executed from notebook (conversion\code\notebooks\dev notebook final v10.ipynb) |
| Unzip ePI document ZIP file to work directory. |

|  |
| --- |
| **Step-2**  **Create Style Rule Book For a specific Language code and Document Type**   1. Check if the style rulebook already exists and load it. 2. Else, create a new default style rulebook for the specific language and document type. |
| **StyleRulesDictionary (conversion\code\parse\rulebook\rulebook.py)** |
| \_\_init\_\_ () |

|  |
| --- |
| **Step-3**  **Convert HTML To JSON along with style based heading level**   1. Convert all images in the document to base64. 2. Read the html document file. 3. Parse the html read previously using beautiful soup. 4. For each element in the soup, prepare a dictionary containing element details, style features and heading level based on style. 5. For assigning levels to each element, style rulebook generated previously is used. 6. All the element dictionaries are stored as a list and are written to an outputJson file. |
| **ParserExtractor (conversion\code\parse\extractor\parser.py)** |
| \_\_init\_\_()  convertImgToBase64( input\_filename)  createPIJsonFromHTML(input\_filepath, output\_filepath, style\_filepath, img\_base64\_dict) |

|  |
| --- |
| **Optional Step (Only For CAP)**  **Split Json file into 4 separate Jsons for different document types.**   1. Create the partitionedJSONs folder in the working root folder if doesn’t exist. 2. Read the OutputJson File generated in the previous step into a pandas dataframe 3. Traverse each row of the pandas dataframe and identity which row’s text matches with the top level Qrd Headings based on jaro winkler similarity. 4. Break the dataframe based on these rows which act as the start of a new section for splitting the document 5. Save each individual dataframe after splitting in the partitionedJSONs folder with the document type appended in the file name. |
| **DocTypePartitioner (conversion\code\htmlDocTypePartitioner\partition.py)** |
| partitionHtmls(qrdkeys, path\_json) |

|  |
| --- |
| **Step-4**  **Get the stop word language keyword using language code**   1. Read the documentTypeNames.json json file 2. Extract the NLTK corpus stop word language keyword for the specific language code of the document being processed. |
| **DocumentTypeNames (conversion\code\languageInfo\documentTypeNames\documentTypeNames.py)** |
| Init()  extractStopWordLanguage() |

|  |
| --- |
| **Step-5**  **Perform heading extraction and validation**   1. Load the partitioned json file (from html) as pandas dataframe 2. Create the failed/missed heading file handler 3. Load the Qrd canonical model data into pandas dataframe and filter specific section as per document 4. Load the match rulebook json file and extract section specific to a language. 5. Traverse through each row in the partitioned dataframe and extract all the rows which are matching with the headings in the Qrd canonical model (step – 5a) and store them as python collection. (Double for loop implemented) 6. In this, the text in html dataframe once matched with a heading in Qrd template is validated across the Qrd template flow and expected scope of headings (step – 5b). 7. Identify the repetition of the sub sections. 8. Check if any mandatory heading got missed in the above process and write such heading into missed heading file. 9. Return the above collection and the original html dataframe (partitioned json) |
| **MatchDocument (conversion\code\match\matchDocument\matchDocument.py)** |
| matchHtmlHeaddingsWithQrd() |

|  |
| --- |
| **Step-5a**  **Match HTML text with QRD Headings**   1. Take text from the document and heading text from Qrd template as input. 2. Preprocess both the text strings (Remove extra spaces) 3. Remove all the stop words from both text strings. 4. Based on the Qrd heading being matched (for special cases using heading number), and its number of words, extract specific section of the match rulebook 5. Find similarity scores of three different types between these two text strings (fuzzy wuzzy, jaro winkler and damerau levenshteins distance 6. Based on the threshholds present in the rules, check all the similarity scores are above the threshold or not 7. If yes for all three , return match is true 8. Else, check if the weighted ratio in fuzzy wuzzy (case insensitive) is above its threshold. If yes, match the texts in lowercase else return false. |
| **MatchStrings (conversion\code\match\matchStrings\matchStrings.py)** |
| getTrueLength(self, str\_)  matchStrings(self, textOriginal, textToMatch,qrdRowHeadingId, avoidLowerCaseMatch=False) |

|  |
| --- |
| **Step-5b**  **Validate Match**  This class is used to validate the Qrd heading found in the html document dataframe based on the previous heading found and validated.   1. For each match found, the validateMatch function takes the several input like current qrd heading row, previous qrd heading row found , previous L1 and L2 headings found, if current heading is from top heading or bottom headings and python collection of heading matched validated so far. 2. In this function we calculate the   previousHeadingRow i.e the previous heading in Level 0,1,2 for the currentHeadingRow with verifying that it exists in the currently parsed headings.  previousHeadingRowQrd i.e the previous heading in Level 0,1,2 for the currentHeadingRow without verfying if the already found or not.   1. Process flow :-  * Check if the previous heading found is same as current heading found , if yes return validated as true. * Check if the current heading found is same as the previous L1 heading found , if yes return true * Check if the current heading found is same as the previous L2 heading found , if yes return true * There are some exceptions in the previous two steps for example the Package leaflet heading don’t repeat. * If previous heading found is null, return true as current heading found will be the first heading * If previous heading found is equal to previousHeadingRow (calculated from current heading found), return True ,else return false. |
| **ValidateMatch (conversion\code\match\validateMatch\validateMatch.py)** |
| validateMatch(currentHeadingRow, previousHeadingRowFound,  previousH1HeadingRowFound, previousH2HeadingRowFound,  dfQrd,  collectionFoundHeadings,  currentHeadingIsTop,  previousHeadingIsBottom) |

|  |
| --- |
| **Step-6**  **Get document annotation information from SPOR API**  This class is used for extracting following details from SPOR API using Marketing authorization.   1. Extract MANs using the dfHtml (pandas dataframe created using outputJson/partitionedJson (created from HTML)) 2. For each MANs from Regulated Authorization API extract   - Author value  - Medicinal Product Definition Ids (Or Packaged Product Definition IDs)  - Optional Step :- From Packaged product extract medicinal product definition   1. For each medicinal product, from its API output extract   - product name  - Administrable prodcut id  - Using Administrable product id extract ingredient id  - Using ingredient api output extract substance id  - Using substance api output extract active substance   1. Collect all the above details for each MAN and return unique values |
| DocumentAnnotation (conversion\code\documentAnnotation\documentAnnotation.py) |
| processRegulatedAuthorization(authorizationIdentifier) |

|  |
| --- |
| **Step-7**  **Extract content between Heading**   1. Load the partitioned json into pandas dataframe. This is used to extract the content of the headings 2. Load the headings collection as pandas dataframe. 3. For each heading in the collection dataframe add two new columns   ‘Text’ :- contains the concatenated text under the heading.  ‘Html\_betw’ :- contains the combined html dom elements of the content under the heading   1. It ignores children of DOM elements with HasBorder and IsULTag true inorder to avoid repetition of dom elements |
| **DataBetweenHeadingsExtractor (conversion\code\extractContentBetweenHeadings\dataBetweenHeadingsExtractor.py)** |
| extractContentBetweenHeadings(input\_filename) |

|  |
| --- |
| **Step-8**  **Read document type code from listBundleDocumentTypeCodes json file** |
| **-** |
| - |
| - |

|  |
| --- |
| **Step-9**  **Generate document bundle xml using Jinja template**   1. Sets system recursion limit to 100000. 2. Creates “fhir\_messages” folder in the working root folder, if it doesn’t exist. 3. Initializes jinja2.FileSystemLoader object, jinja2.Environment object and assigns the path of the jinja template to these. 4. Creates an object named “xml\_bundle\_data” containing all meta data of the template This includes the PMS/OMS annotation data and the style information read from the stylesheet text file from the OutputJSON. 5. Creates a tree “id\_dict\_list” based on the “id” column in the dataframe. This is used to populate the sections tag of the FHIR XML. 6. For every element in the dataframe, checks if there are any children with <img> and extracts the data uri and type of the image. The extracted information is added to a dictionary “img\_ref\_dict” which is used to populate the Contained tag of the FHIR XML. 7. xml\_bundle\_data, img\_ref\_dict, id\_dict\_list and the root of the tree in id\_dict\_list are sent to the jinja template renderer to create the FHIR message. 8. The xml is written onto the "fhir\_messages" folder. |
| **FhirXmlGenerator (conversion\code\fhirXmlGenerator\fhirXmlGenerator.py)** |
| generateXml(df, xml\_file\_name) |

|  |
| --- |
| **Step-10**  **Submit document bundle to FHIR server**   1. Submit the document bundle to FHIR server using python requests library. |
| FhirService (conversion\code\fhirService\fhirService.py) |
| submitFhirXml() |

|  |
| --- |
| **Step-11**  **Add or update list document**  This class is used to update or add the list bundle for a particular medicine.   1. Using the MANs extracted from the document in the step – 6 2. Get the list bundle from the FHIR server using the above MANs and in case of NAP and package leaflet doc type get the list using the med name. 3. If more than 1 list bundle is returned, raise an error 4. If not present, load the default creation template for list bundle. 5. Add MAN in the identifier 6. Add lowercase name of the medicine in the identifier 7. Add MAH in the extension 8. Add domain in the extension 9. Add active substance in the extension 10. Update the document bundle reference in the list 11. Update or add the above list bundle |
| ListBundleHandler (conversion\code\listBundle\addAndUpdateListBundle\addAndUpdateListBundle.py) |
| submitFhirXml() |