

The Parrot user interface for annotating documents is a frontend Javascript-heavy interface accessible in the browser. This document serves to explain the core logic behind the labelling of the documents, starting from the user's input in highlighting parts of the HTML document, to the JSON output that determines the character positions (start, end) for each highlighted text in the HTML-escaped raw document text.

This documentation does not cover generic operations in Parrot such as uploading files, connection to DB, Flask integration etc. This documentation also does not cover advanced features such as pre-labelling, merging & copying jobs, adding parameters to annotated texts, etc.

**1. Labelling Text in Javascript**

The interface where the user starts to label is the HTML render of the raw document text page. When the user clicks a word, the word will be 'marked' and highlighted as shown:



Note:

* Click and drag for finer control of the highligting is possible in cases where the user does not want to mark the whole word.
* Whitespaces before and after the marked text are not included (and not highlighted).

The function highlight() is the onclick handler on the document text that captures the users selection input (click and drag). Once onkeyup is detected, the function registers the start and end of the document text selected by the user. Note that the start and end positions here are still irrelevant as this entire text is in HTML and is not in the raw text format required for the output.

**highlight()**

|  |
| --- |
| function highlight() {      var selection = window.getSelection();        if (selection.anchorNode === null) {          // Do nothing if nothing is selected          return null      }        var selection\_range = selection.getRangeAt(0);        var start\_offset = selection\_range.startOffset;      var end\_offset = selection\_range.endOffset;        elems\_in\_selection = get\_all\_elements\_in\_selection(selection);      if (elems\_in\_selection.length > 0) {          // When there are existing elems in the selected text.          // Do nothing and remove current highlighting          selection.removeAllRanges();          return null      }      mark\_content(selection);        selection.removeAllRanges();        return null  } |

**mark\_content()**

|  |
| --- |
| function mark\_content(selection) {      // Given a Selection object, create mark tag around text      // Then add the label to the nav bar          var range = selection.getRangeAt(0);      range = clean\_range(selection, range);      var start\_offset = range.startOffset;      var end\_offset = range.endOffset;        // If after cleaning there are no characters other than whitespace      // Do nothing      if (selection.toString().trim() === "") {          return null      }      var mark = document.createElement('mark');        var mark\_id = createUUID();      mark.setAttribute("id", mark\_id);      mark.setAttribute("label", LABEL);      mark.classList.add("highlight");      mark.style.backgroundColor = LABEL\_COLOR;        mark.appendChild(range.extractContents());      range.insertNode(mark);      ATTRIBUTES[mark\_id] = {          "spelling":""      }; // Initialize attributes        // Set mousedown event      mark.onmousedown = function(e) {          markMousedown(this, e);      };        append\_label\_to\_nav(selection.toString(), LABEL, mark\_id);    } |

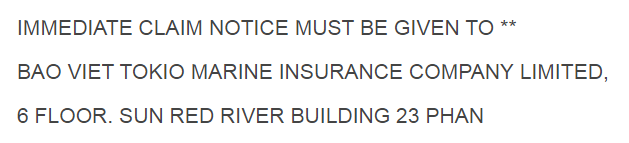
Within the highlight() function, mark\_content() is called to encapsulate the highlighted text in the <mark></mark> HTML tag which holds the following data regarding the highlighted text:

* id : A UUID4 identifier is generated and assigned into the <mark> tag
* label : The label of the highlighted text. This is selected by the user before deciding to highlight the target text.
* highlight : Adds the highlight CSS class to the marked text that gives the text the highlighted color and style according to the label  
  In addition, mark\_content() adds a mousedown attribute that will allow the user to remove the highlight when it is clicked again.

For example:

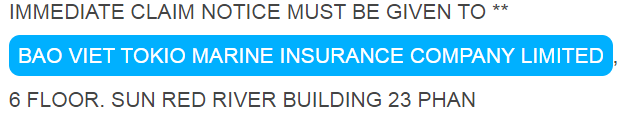
1. **HTML text before highlighting:**

BAO VIET TOKIO MARINE INSURANCE COMPANY LIMITED

User sees:  


1. **HTML text after highlighting:**

|  |
| --- |
| <mark id="cf78eff9e5134e3f5334e095633a5286" label="agent\_at\_destination" class="highlight" style="background-color: rgb(25, 118, 210);">BAO VIET TOKIO MARINE INSURANCE COMPANY LIMITED</mark> |

User sees:  


markMousedown() is the mousedown handler for the <mark> tag which will unravel the inner contents of the tag and merge it back to the innerHTML content of the parent div. This will ensure the removing of the mark will return the HTML to the original state before being highlighted.

|  |
| --- |
| function markMousedown(elem, e) {      // This is the default mousedown event handler for the <mark> elements      // Should be handled by the <mark> elements onmousedown event      // parameter `e` refers to the event of onmousedown      var e = e || window.event;      if (e.which === 3) {          // Detect right click event.          // Disallow unmarking with right click          // Do something here            if (elem.classList.contains('pre-label')) {              // Right click unwraps and removes <mark> tag if it is a pre-label class              unwrap(document.getElementById(elem.getAttribute('id')));          } else {              // For normal <mark> tags, right click opens up the spelling overlay              initSpellingContainer(elem.getAttribute("id"));              spellingOverlayOn();          }          } else {          // Detect left click events here            if (elem.classList.contains('pre-label')) {              // If it is a pre-label that has been clicked              // Convert it to normal label              // - Add it to side nav              // - Update ATTRIBUTES for this label              elem.classList.remove('pre-label');              append\_label\_to\_nav(                  elem.innerText,                  elem.getAttribute('label'),                  elem.getAttribute('id')              );              ATTRIBUTES[elem.getAttribute('id')] = {                  "spelling":""              }; // Initialize attributes              elem.style.color = null;              elem.style.borderColor =null;              elem.style.backgroundColor = LABEL\_PALETTE[elem.getAttribute('label')];                e.preventDefault();            } else {              unwrap(document.getElementById(elem.getAttribute('id')));              remove\_label\_from\_nav(elem.getAttribute("id"));              delete ATTRIBUTES[elem.getAttribute('id')];          }      }  }    function unwrap(elem) {      // Given inner element, remove surrounding tags      // while retaining content and merging with outer parent.        var parent = elem.parentElement;      parent.innerHTML = parent.innerHTML.replace(          elem.outerHTML,          elem.innerHTML      )        // The unwrap mousdown event is removed for any other mark      // objects in the same parent element.      // Need to set the mousedown event again.        other\_marks = parent.querySelectorAll("mark")      for (i=0; i<other\_marks.length; i++) {          other\_marks[i].onmousedown = function(e) {              markMousedown(this, e);          };      }  } |

It is important to note that the unraveling of the <mark> tag and the subsequnt merge into the parent innerHTML will reset all mousedown events in all other <mark> tags in the page. The unwrap() function call handles that by re-introducing the mousedown events to the remaining <mark> tags again.

**2.Handling State**

The itermediary state of the working document is saved locally in the application folder /jobs with each document saved within a job folder that is named after its UUID4 identifier. E.g., /jobs/01b7f33cb5ed48c9996c9047b76eefd4/. The document was first split into its pages where each page's HTML working file is saved within its jobs folder. The pages were initially split by the identifying character \x0c character during the start of the Parrot job document upload.

After annotating each page, the user needs to click the save button or click the next page icon, both of which will initiate the save\_page\_api API call into Flask. This saves a snapshot of the HTML page document and stores it locally in a text file.

**Frontend JS XMLHttpRequest API Call**

|  |
| --- |
| function savePage(job\_id, page\_no) {      // Saves current text-blob      var xhttp = new XMLHttpRequest();    var saveButton = document.getElementById("save-page");    saveButton.classList.add("saving");    saveButton.innerText = "..saving";    saveButton.disabled = true;      xhttp.onreadystatechange = function() {      if (this.readyState == 4 && this.status == 200) {          saveButton.classList.remove("saving");          saveButton.innerText = "Save ";          saveButton.disabled = false;          var response = JSON.parse(this.responseText);          alert("Document saved successfully");        }    };    // Save page additional details in ATTRIBUTES    ATTRIBUTES["page\_class"] = getPageClass();    ATTRIBUTES["page\_remarks"] = getPageRemarks();    data = {      "job\_id": job\_id,      "page\_no": page\_no.toString(),      "blob": document.getElementById('text-blob-p').innerHTML.trim(),      "attributes": ATTRIBUTES,    }    xhttp.open("POST", "/api/save", true);    xhttp.setRequestHeader("Content-Type", "application/json;charset=UTF-8");    xhttp.send(JSON.stringify(data));  } |

**Backend Python Flask API handler**

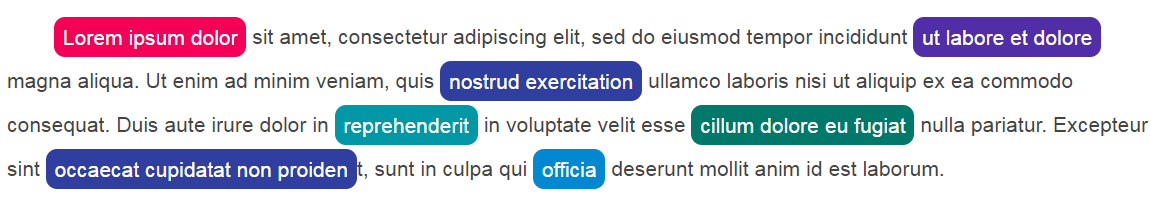
|  |
| --- |
| @api.route('/api/save', methods=['POST'])  def save\_page\_api():      """ POST endpoint to save page when user changes page, completes job or clicks      the save button.        Saving overwrites the temp page content file in the job folder      with current html blob from the annotator view.        Example      -------          Required POST json example          {              "job\_id":"3d3f678d37194e3dab4927711a53189b",              "blob": "raw\_html\_content\_here....",              "page\_no":1          }      """        data = request.get\_json()      job\_id = data['job\_id']      content = data['blob']      page\_no = data['page\_no']      attributes = data['attributes']      temp\_file\_name = f"{job\_id}-temp-{page\_no}.txt"      attributes\_file\_name = f"{job\_id}-attributes-{page\_no}.json"      job\_folder = os.path.join(JOBS\_DIR, job\_id)        with open(os.path.join(job\_folder, temp\_file\_name), 'w', encoding='utf-8') as f:          f.write(content)        with open(os.path.join(job\_folder, attributes\_file\_name), 'w', encoding='utf-8') as f:          f.write(json.dumps(attributes))        results = get\_annotations(content, job\_id, page\_no)      check\_annotation\_results(          results['content'],          results['annotations']      )        # Open raw page file to compare contents for integrity issue      raw\_file\_name = f"{job\_id}-raw-{page\_no}.txt"      with open(os.path.join(job\_folder, raw\_file\_name), 'r', encoding='utf-8') as f:          raw\_content = f.read()        # Compare contents from user upload vs after annotation      check\_content\_integrity(results['content'], raw\_content)        return make\_response(jsonify({"message":"saving successful", \*\*results}), 200) |

The above two functions in JS and Python respectively shows the connection between frontend and backend, and the subsequent handling of the intermediate HTML file with the labels saved into the jobs folder.

**3.Label Positioning**

The crux of the Parrot annotator logic is the efficient determination of the start and end character positions for each label highlighted by the user. The problem to solve here is that the working document is in HTML format which contains all forms of HTML tags inserted into the document content. This causes the <mark> positionings of the labels to be incorrect as the total character count is larger than the original raw text form. Note that storing the intermediate content as HTML is inevitable as this is the format the user has to interface with the application.  
  
Once the page is saved by the save\_page\_api handler, the function get\_annotations() handles the conversion of the HTML page to its original raw text form while determining the accurate label positions saved.

**Example User Labelling**



**HTML Blob of User's Labelling**

<mark id="a3db9fe36059b6c99896a6a8ebc603c9" label="country\_of\_orgin" class="highlight" style="background-color: rgb(245, 0, 87);">Lorem ipsum dolor</mark> sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt <mark id="64b149c611ddacc61833e41e793b6099" label="drawee" class="highlight" style="background-color: rgb(81, 45, 168);">ut labore et dolore</mark> magna aliqua. Ut enim ad minim veniam, quis <mark id="a4e705c5cd3b305c2b182c5a8476a3ca" label="issuing\_bank" class="highlight" style="background-color: rgb(48, 63, 159);">nostrud exercitation</mark> ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in <mark id="d5fa6123ab50a0fdfe9ed4c4afc3014d" label="vessel\_appln" class="highlight" style="background-color: rgb(0, 151, 167);">reprehenderit</mark> in voluptate velit esse <mark id="57f00eb801cdbd1527af329f78a1c0be" label="port\_of\_loading\_appln" class="highlight" style="background-color: rgb(0, 121, 107);">cillum dolore eu fugiat</mark> nulla pariatur. Excepteur sint <mark id="d5fe3a566910f5f1dbc04da986fa2a16" label="issuing\_bank" class="highlight" style="background-color: rgb(48, 63, 159);">occaecat cupidatat non proiden</mark>t, sunt in culpa qui <mark id="a26c224cc20aa7cf7c7451b7b2c79f0a" label="applicant" class="highlight" style="background-color: rgb(2, 136, 209);">officia</mark> deserunt mollit anim id est laborum.

|  |
| --- |
|  |

**Desired Output: HTML-Escaped Original Text + Label Positioning in JSON**

* JSON containing list of annotations with the start and end character positions in the tuple pos.
* content containing original HTML-escaped string text.

{

'annotations': [

{'text': 'Lorem ipsum dolor', 'pos': (0, 17), 'label': 'country\_of\_orgin', 'spelling': ''},

{'text': 'ut labore et dolore', 'pos': (90, 109), 'label': 'drawee', 'spelling': ''},

{'text': 'nostrud exercitation', 'pos': (154, 174), 'label': 'issuing\_bank', 'spelling': ''},

{'text': 'reprehenderit', 'pos': (257, 270), 'label': 'vessel\_appln', 'spelling': ''},

{'text': 'cillum dolore eu fugiat', 'pos': (295, 318), 'label': 'port\_of\_loading\_appln', 'spelling':''},

{'text': 'occaecat cupidatat non proiden', 'pos': (350, 380), 'label': 'issuing\_bank', 'spelling': ''}, {'text': 'officia', 'pos': (401, 408), 'label': 'applicant', 'spelling': ''}

],

'content': 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.'

}

**get\_annotations()**

|  |
| --- |
| def get\_annotations(html\_raw\_content, job\_id, page):      """ Given string content of html saved output from frontent      Return parsed (no-HTML) blob of text together with annotations in a dictionary      Uses Pure REGEX - No BeautifulSoup        Args          html\_raw\_content (str): Raw HTML content (utf-8) submitted from frontend          job\_id (str): UUID4 string representing the job ID          page (int): Page number of current html content in document        Returns          results (dict): Dictionary of values containing parsed (no-HTML) blob of text                          With list of annotations of locations and labels of substrings        """      job\_folder = os.path.join(JOBS\_DIR, job\_id)        # Find all <mark>...</mark> tags before escaping HTML      marks = re.findall("<mark [\s\S]\*?</mark>", html\_raw\_content)      annotations = []        # Need to unescape HTML and re-compile page with no <mark> tags      # Must be equivalent to raw page content      content\_no\_marks = html.unescape(html\_raw\_content)        # Get saved attributes      attributes\_file\_name = f"{job\_id}-attributes-{page}.json"      try:          with open(os.path.join(job\_folder, attributes\_file\_name), 'r') as f:              attributes = json.loads(f.read())              if not attributes:                  attributes = {}      except Exception as e:          logging.debug("!! Error getting attributes for ", job\_id, " ", page)          logging.debug(e)          attributes = {}        for i, mark in enumerate(marks):            outer\_html = mark          inner\_text = html.unescape(re.findall('<mark [\s\S]\*?">([\s\S]\*?)</mark>', mark)[0])          label = re.findall('<mark [\s\S]\*? label="([\s\S]\*?)"[\s\S]\*?>', mark)[0]          classes = re.findall('<mark [\s\S]\*? class="([\s\S]\*?)"[\s\S]\*?>', mark)[0].split(" ")          mark\_id = re.findall('<mark id="([\s\S]\*?)"[\s\S]\*?>', mark)[0]          spelling = ""            mark\_attributes = attributes.get(mark\_id)          if mark\_attributes:              spelling =  mark\_attributes.get("spelling", "")            start\_pos = content\_no\_marks.find(              html.unescape(outer\_html)          ) # content\_no\_marks is already unescaped. So need          # to find position from unescaped outer\_html            end\_pos = start\_pos + len(inner\_text)            content\_no\_marks = content\_no\_marks.replace(              html.unescape(outer\_html),              inner\_text          )            if "pre-label" not in classes:              # Only save this annotation              # if user removed the "pre-label" class in the              # <mark> tag                annotations.append({                  "text": inner\_text,                  "pos": (start\_pos, end\_pos),                  "label": label,                  "spelling": spelling,                  # "outer\_html":mark              })        return {          "annotations":annotations,          "content":content\_no\_marks,      } |

The algorithm works in the following manner:

1. Given the raw HTML text page, extract out all <mark>...</mark> content using the regex pattern <mark [\s\S]\*?</mark>. This will give a list marks of <mark>...</mark> html strings in *order of their appearance* in the HTML document.
2. Iterate through each <mark>...</mark> string in marks to do the following:
   1. Extract the relevant information such as the text string in the mark inner\_text, the label type, and the UUID mark\_id. These extractions are done solely through regex patterns applied to the mark string.
   2. Find the position start\_pos in the raw HTML text that is the first match the of string <mark>...</mark>. The match will be unique in the document due to the existence of the UUID in the <mark> tag.
   3. Next calculate the end position of the label by adding the length of the inner\_text to start\_pos.
   4. Replace the HTML markup <mark>...</mark> with the inner\_text in the raw HTML document stored temporarily as content\_no\_marks. This essentially removes the HTML tags and progressively converts the document back to its original raw text form.
3. Iteratively, step 2 will convert the HTML document content\_no\_marks into its raw string format by the end of the loop. The start\_pos, end\_pos for each label is stored in the annotations list and the final content\_no\_marks is eventually saved and returned from the function.

**check\_annotation\_results()**

Once get\_annotations() returns a result, the function check\_annotation\_results() can be called to validate the label positions to ensure that the calculated start and end positions of the extracted text correspoind to the exact text within the raw string document page.  
  
For example the extracted text 'World' with the calculated start and end positions (6, 10) should correspond to the text 'World' in the content Hello World. Failing which the function will throw an assertion error.

|  |
| --- |
| def check\_annotation\_results(content, annotations):      """ Given parsed HTML content and annotations generated from      `generate\_annotations`, check that the positions of the annotations      correspond to the same text positions in the content blob        """      for anno in annotations:          assert(content[anno['pos'][0]:anno['pos'][1]] == anno['text']) |

**Summary**

The process described in this document underpins the main purpose of Parrot, which is to allow users to easily annotate a document to derive the exact positions and labels of the annotations within the text. The steps to produce these annotations are replicated for each page in the document and once the job is complete (initiated by the user), it will be compiled into a JSON object to be inserted into a MongoDB database.