Documentation - classes and functions

CONTENT.JS

`readClipboardData()`:

- Checks if the extension is enabled (`enabled` key in `chrome.storage.local`).
- Reads clipboard data using `navigator.clipboard.read()` and processes it.
- Handles image types (e.g., 'image/tiff') by calling `setClipboardImage()` if new data is detected.
- Handles text data (type 'text/plain') by calling `setClipboardText()` if new data is detected.

`setClipboardImage(imageBlob)`:

- Reads image data from the clipboard and converts it to a data URL using `FileReader`.
- Ensures the image is unique before adding it to the `imageList` stored in `chrome.storage.local`.
- Keeps the `imageList` size within the `_maxListSize` limit.

`setClipboardText(clipText)`:

- Retrieves various clipboard-related lists from `chrome.storage.sync` (e.g., `list`, `listURL`, `originalList`, `summarizedList`, `citationList`).
- Checks for duplicates and maintains the size of these lists within `_maxListSize`.
- Adds new clipboard text and its related data (e.g., current URL) to these lists and updates the storage.

Event Listeners:

- `window.addEventListener('mouseout')`: Starts a periodic interval (`readClipboardData()` every 2000ms) when the mouse leaves the window.
- `window.addEventListener('mouseover')`: Stops the interval set by `mouseout`.
- `window.addEventListener('copy')`: Invokes `readClipboardData()` when a copy event is detected.
- `document.addEventListener('visibilitychange')`: Clears or sets the interval depending on the visibility state of the document.

POPUP.JS

`checkMode()`: Toggles between dark mode and light mode.
`darkmodeOn()`: Enables dark mode by adding a CSS class to elements.
`darkmodeOFF()`: Disables dark mode by removing the CSS class.
`getClipboardText()`: Retrieves clipboard text data from storage and updates the clipboard list in the popup.
`getClipboardImages()`: Fetches stored images and adds them to the clipboard list.
`addClipboardImageItem(imageDataUrl)`: Adds an image entry to the clipboard list and sets up event listeners for actions (copy, move, delete).
`deleteImageItem(imageDataUrl)`: Deletes an image entry from the storage and updates the display.
`moveImageItemUp(imageDataUrl)` / `moveImageItemDown(imageDataUrl)`: Moves an image entry up or down in the list.
`getThumbnail(textContent)`: Identifies whether a text is a URL or YouTube link and returns thumbnail details.
`addClipboardListItem(text, item_color)`: Adds a text entry to the clipboard list with event listeners for editing, deleting, moving, and highlighting.
`downloadClipboardTextAsDoc()`: Downloads clipboard text data as a Word document.
`downloadClipboardTextAsCsv()`: Downloads clipboard text data as a CSV file.
`deleteAllText()`: Clears all text entries from the clipboard storage and UI.
`searchClipboardText()`: Filters the displayed clipboard list based on a search term.
`enableHighlightMode(element)`: Enables a mode for highlighting selected text within a clipboard entry.
`saveHighlight(originalText, highlightedHTML)`: Saves highlighted text to Chrome storage.
`applySavedHighlights(element, originalText)`: Applies previously saved highlights when loading clipboard items.
`showSnackbar(message)`: Displays a notification message at the bottom of the popup.
`doDjangoCall(type, url, data, callback)`: Makes an AJAX request to a specified URL and handles the response.
`deleteElem(text)`: Deletes a specific text entry from the clipboard storage and updates related data.

Event Listeners:

- Toggle buttons for enabling/disabling clipboard tracking and dark mode.
- Click events for adding, merging, editing, and deleting clipboard entries.
- Input events for resizing text areas and highlighting search matches.

BACKGROUND.JS

The `background.js` file serves as the background script for the SimplyClip Chrome extension.

`chrome.contextMenus.create()`:

- **Purpose**: Creates a context menu item titled "Save Image to SimplyClip" that appears when a user right-clicks on an image.
- **Details**:
  - `title`: Sets the label of the context menu item.
  - `contexts`: Specifies the type of content that the menu appears on (in this case, images).
  - `id`: A unique identifier for the context menu item.

`chrome.contextMenus.onClicked.addListener(function (info, tab))`:

- **Purpose**: Listens for a click event on the context menu item created above.
- **Parameters**:
  - `info`: Contains details about the context menu click, such as the ID of the menu item and the source URL of the image.
  - `tab`: Provides information about the tab where the menu item was clicked.
- **Functionality**:
  - Checks if the `menuItemId` is "save-image-to-simplyclip".
  - If true, extracts the `srcUrl` (the image URL) from the `info` object.
  - Fetches the image data from the provided URL and converts it to a `Blob` for further processing.

`fetch(imageUrl).then((response) => response.blob()).then((blob) => { ... })`:

- **Purpose**: Fetches the image data from the URL and converts it into a `Blob` for further handling.
- **Details**:
  - Uses the `fetch` API to get the image data from the URL.
  - Converts the response to a `Blob` format to be processed by the `FileReader`.

`const reader = new FileReader();` and `reader.onload = function (e) { ... };`:

- **Purpose**: Reads the `Blob` data and converts it into a data URL format.
- **Details**:
  - `FileReader` reads the `Blob` and triggers the `onload` event when the reading is complete.
  - The `onload` function captures the data URL (`e.target.result`) of the image.

`chrome.storage.local.get(["imageList"], function (result) { ... })`:

- **Purpose**: Retrieves the current list of saved images from `chrome.storage.local`.
- **Details**:
  - If no `imageList` exists, initializes it as an empty array.
  - Checks if the current image data URL is already in the list to avoid duplicates.

`imageList.unshift(imageDataUrl); chrome.storage.local.set({ imageList: imageList }, () => { ... });`:

- **Purpose**: Adds the new image data URL to the beginning of the `imageList` array and updates the storage.
- **Details**:
  - `unshift` ensures the new image appears at the top of the list.
  - `chrome.storage.local.set` saves the updated `imageList` back to Chrome's local storage.

MANAGE.PY

This is a standard utility script for all Django projects.

1. **`main()` Function**:
   - **Purpose**: Acts as the entry point for executing Django administrative tasks.
   - **Details**:
     - Sets the default environment variable `DJANGO_SETTINGS_MODULE` to point to the Django project's settings module (`simplyclip_backend.settings` in this case).
     - Imports and executes `execute_from_command_line()` from `django.core.management`, which handles command-line commands like `runserver`, `migrate`, `startapp`, etc.
     - If Django is not installed or cannot be imported, an `ImportError` is raised with a detailed message guiding the user to check their installation and environment setup.
2. **`if __name__ == "__main__":` Block**:
   - **Purpose**: Ensures that the `main()` function is called only when `manage.py` is run as a script (i.e., not when it is imported as a module).
   - **Details**: Executes the `main()` function, which processes any command-line arguments passed and runs the corresponding Django command.

- **`os.environ.setdefault("DJANGO_SETTINGS_MODULE", "simplyclip_backend.settings")`**:
    - Sets up the environment to use the Django settings module for the project.
- **`execute_from_command_line(sys.argv)`**:
    - Takes command-line arguments (e.g., `runserver`, `migrate`) and executes the corresponding administrative task.

CITATION.PY

The `citation.py` file is designed to generate formatted citations for given text input using the `habanero` library, which interfaces with the Crossref API. It supports multiple citation formats and attempts to create citations based on DOI (Digital Object Identifier) or by searching for relevant articles when a DOI is not directly provided.

1. **`class CitationError(Exception)`**:
    - **Purpose**: Defines a custom exception type called `CitationError`.
    - **Details**: This custom exception can be raised when citation generation encounters a specific failure, such as not finding matching articles.
2. **`createCitation(text)`**:
    - **Purpose**: Generates citations in multiple styles for a given input text, either interpreted as a DOI or searched for as an article.
    - **Parameters**:
        - `text`: The input text, which can be a DOI or a search query for an article.
    - **Details**:
        - **Supported Citation Types**: APA, BibTeX, Chicago Author-Date, MLA, and Vancouver.
        - **Function Workflow**:
            - **Attempt DOI-Based Citation**:
                - Tries to generate citations by interpreting `text` as a DOI.
                - Uses `habanero.cn.content_negotiation()` to fetch citation data for the given DOI in various formats.
            - **Fallback to Article Search**:
                - If DOI-based citation fails, it searches for articles using `habanero.Crossref()` and the provided `text` as a query.
                - If an article is found, it retrieves the DOI and regenerates the citation using the discovered DOI.
        - **Error Handling**:
            - If both DOI interpretation and article search fail, returns an error message stating that the citation generation was unsuccessful.

    - **Return Value**:
        - Returns a string containing formatted citations in multiple styles or an error message if the process fails.

Code Details

- **`habanero.cn.content_negotiation()`**: Fetches citation data using Crossref's content negotiation.
- **`works.works(query=text, limit=1)`**: Searches for an article that matches the given text and retrieves metadata, including the DOI.
- **Error Messages and Logging**:
    - The function prints informative messages to the console about the status of citation generation, including failures and successful completions.

CITATIONTEST1.PY

The purpose of `citationTest1.py` is to ensure that the `createCitation` function in `citation.py` behaves as expected. It does so by using unit tests with mock objects to simulate the behavior of external dependencies, specifically the `habanero` library's `content_negotiation` function.

Explanation of Functions and Code

1. **`import unittest`**:
    - Imports Python's built-in `unittest` framework for creating and running test cases.
2. **`from unittest.mock import patch`**:
    - Imports the `patch` function from `unittest.mock`, which is used to replace parts of the system under test with mock objects during test runs.
3. **`import citation`**:
    - Imports the `citation` module, which contains the `createCitation` function being tested.
4. **`class GenerateCitationTest(unittest.TestCase)`**:
    - Defines a test case class that inherits from `unittest.TestCase`, allowing it to contain test methods.
5. **`@patch('habanero.cn.content_negotiation')`**:
    - Decorates the `test_generate_citation` method, replacing `habanero.cn.content_negotiation` with a mock object to simulate its behavior during the test.
    - This avoids calling the actual external API and provides controlled responses.

6. **`def test_generate_citation(self, mock_content_negotiation)`**:
   - A test method that evaluates the `createCitation` function's behavior.
   - **Parameters**:
     1. `self`: The instance of the test class.
     2. `mock_content_negotiation`: The mock object for the `habanero.cn.content_negotiation` function.
   - **Test Scenarios**:
     1. **Normal Case**:
        - The mock object is set to return `'Mocked Citation'` for any call.
        - The test checks if `createCitation` produces a formatted output matching the expected citation styles with the mock response.
     2. **Exception Case**:
        - The mock object is set to raise an exception when called (`mock_content_negotiation.side_effect`).
        - The test verifies that `createCitation` raises a `CitationError` when an exception occurs.
7. **`if __name__ == '__main__': unittest.main()`**:
   - Runs the test case when the script is executed directly.
   - `unittest.main()` discovers and runs all test methods defined in the `GenerateCitationTest` class.

SUMMARIZER.PY

The `summarizer.py` file provides a text summarization utility that processes input text to create a summary based on the importance of sentences, determined by word frequencies. The function removes common stopwords, tokenizes the text, and scores sentences based on their content. The highest-scoring sentences are included in the final summary up to a specified maximum word count.

Explanation of Functions and Code

1. **Imports**:
   - **`nltk`**, **`re`**: Used for natural language processing tasks and regular expression operations.
   - **`stopwords`**, **`word_tokenize`**, **`sent_tokenize`** from `nltk`: Provide tools for tokenizing text into words and sentences and removing common stopwords.
   - **`defaultdict`** from `collections`: Used for creating dictionaries with default values.

2. **`generate_summary_v2(text: str, max_words: int = 50)`**:
    - **Purpose**: Creates a summary of the input text, limited to a specified maximum number of words.
    - **Parameters**:
        - `text` (str): The input text to be summarized.
        - `max_words` (int): The maximum number of words allowed in the summary (default is 50).
    - **Returns**: A string containing the summary.
    - **Raises**:
        - `ValueError`: If the input text is empty or `max_words` is less than 1.
        - `RuntimeError`: If issues arise during various stages (e.g., tokenization, word frequency calculation).
3. **Function Workflow**:
    - **Standardization**:
        - Converts the input text to lowercase and removes non-alphanumeric characters.
    - **Stopword Removal and Tokenization**:
        - Tokenizes the text and filters out common stopwords using NLTK's stopword list.
    - **Word Frequency Calculation**:
        - Counts the frequency of each word, normalized by the most frequent word's count.
    - **Sentence Tokenization and Scoring**:
        - Splits the text into sentences and scores them based on the sum of normalized word frequencies of words present in each sentence.
    - **Summary Construction**:
        - Selects and joins the highest-scoring sentences until the maximum word count is reached.
4. **Error Handling**:
    - Comprehensive error handling is present to raise meaningful errors if issues occur at various stages (e.g., tokenization or stopword download issues).

SUMMARIZER_TEST.PY

The script verifies that `generate_summary_v2` behaves as expected under various scenarios, including typical use cases and edge cases. It uses the `unittest` framework with `mock` to replace external dependencies and ensure that the tests focus solely on the function's behavior.

Explanation of Functions and Test Cases

1. **`import unittest`**:
    - Imports the `unittest` module, allowing the creation of test cases.

2. **`from unittest.mock import patch`**:
    - Imports `patch` to mock functions and dependencies for isolated testing.
3. **`from summarizer import generate_summary_v2`**:
    - Imports the `generate_summary_v2` function from `summarizer.py` for testing.
4. **`class SummarizeEndpointTest(unittest.TestCase)`**:
    - Defines a test case class that inherits from `unittest.TestCase`. It contains individual test methods that check different behaviors of the `generate_summary_v2` function.
5. **Test Methods**:
    - **`@patch("habanero.cn.content_negotiation")`**:
        - Decorates each test method to mock the `habanero.cn.content_negotiation` function, ensuring external API calls don't interfere with the tests.
    - **`test_summarize_correct_length(self, mock_generate_summary)`**:
        - **Purpose**: Verifies that the generated summary is a string and that the word count does not exceed the specified `max_words` (50 in this case).
        - **Process**:
            - Mocks the `content_negotiation` function.
            - Calls `generate_summary_v2` with sample text.
            - Checks that the response is a string and the word count is less than or equal to 50.
    - **`test_invalid_text_type(self, mock_content_negotiation)`**:
        - **Purpose**: Ensures that the function raises a `ValueError` when `text` is not a string.
        - **Process**:
            - Calls `generate_summary_v2` with a non-string input (e.g., 123).
            - Asserts that the raised exception matches the expected message.
    - **`test_empty_text(self, mock_content_negotiation)`**:
        - **Purpose**: Verifies that the function raises a `ValueError` when the input text is empty or consists only of whitespace.
        - **Process**:
            - Calls `generate_summary_v2` with a string of spaces.
            - Asserts that the raised exception matches the expected message.
    - **`test_invalid_max_words_type(self, mock_content_negotiation)`**:
        - **Purpose**: Checks that the function raises a `ValueError` when `max_words` is not an integer.
        - **Process**:
            - Calls `generate_summary_v2` with a string value for `max_words`.
            - Asserts that the raised exception matches the expected message.

- **`test_invalid_max_words_value(self, mock_content_negotiation)`**:
    - **Purpose**: Ensures that the function raises a `ValueError` when `max_words` is less than or equal to zero.
    - **Process**:
        - Calls `generate_summary_v2` with `max_words` set to `-1`.
        - Asserts that the raised exception matches the expected message.
- **`test_no_stopwords_found(self, mock_content_negotiation)`**:
    - **Purpose**: Verifies the behavior when the input text has no common words that are filtered out.
    - **Process**:
        - Calls `generate_summary_v2` with a rare or unique word ("Floccinaucinihilipilification").
        - Asserts that the summary is the original word itself, as it is the only content.

VIEWS.PY

The `views.py` file is part of a Django application and contains view functions that handle HTTP requests. The functions provide various services such as text summarization, citation generation, file upload, and fetching files. These views use Django's HTTP response objects to send responses back to the client and use `@csrf_exempt` decorators to bypass CSRF protection for specific endpoints.

Explanation of Functions

1. **`@csrf_exempt def summarize(request, summ_input):`**
    - **Purpose**: Handles HTTP `GET` requests to generate a summary of the provided text input.
    - **Parameters**:
        - `request`: The HTTP request object.
        - `summ_input`: The input text to be summarized.
    - **Process**:
        - Extracts the text from the request.
        - Calls `summarizer.generate_summary_v2()` to create a summary.
        - Returns the summarized text as an `HttpResponse` with `text/plain` content type.
    - **Use Case**: Provides a direct API endpoint for generating a text summary.
2. **`@csrf_exempt def getcitation(request):`**
    - **Purpose**: Handles HTTP `POST` requests to generate citations based on input text.

- **Parameters**:
    - `request`: The HTTP request object.
- **Process**:
    - Extracts `citation_input` from the request's POST data.
    - Calls `citation.createCitation()` to generate the citation.
    - Returns the generated citation as an `HttpResponse` with `text/plain` content type.
- **Error Handling**: Returns an HTTP 400 response if the request method is not `POST`.
- **Use Case**: Allows users to submit text and receive formatted citations in response.

3. **`@csrf_exempt def upload(request):`**
    - **Purpose**: Handles file uploads via HTTP `POST` requests.
    - **Parameters**:
        - `request`: The HTTP request object.
    - **Process**:
        - Checks if the request contains a file under `myfile`.
        - Uses `FileSystemStorage` to save the uploaded file to a `docs/` directory.
        - Prints a success message and returns an `HttpResponse` indicating successful file storage.
    - **Use Case**: Facilitates file uploads to the server, storing them in a predefined directory.
4. **`@csrf_exempt def fetch(request, summary_in):`**
    - **Purpose**: Handles HTTP `GET` requests to package and return files in a ZIP archive.
    - **Parameters**:
        - `request`: The HTTP request object.
        - `summary_in`: A parameter passed to the function (though not directly used in this function).
    - **Process**:
        - Triggers a `pdb.set_trace()` for debugging (typically removed in production).
        - Creates a ZIP file (`docs.zip`) containing all files from the `docs/` directory.
        - Uses `FileResponse` to return the ZIP file to the client.
    - **Use Case**: Provides a way to download all documents from the server as a single ZIP archive.
    - **Output**: Returns a `FileResponse` to enable file download.

SETTINGS.PY

The `settings.py` file is a core part of a Django project. It contains the configuration for the Django application, defining how the project behaves, the middleware it uses, installed applications, database settings, and more. This file is essential for the initialization and functioning of a Django application.

Key Sections and Their Purposes

1. **Project Paths and Base Directory**:
   - **`BASE_DIR`**: Defines the base directory of the project using `Path` from the `pathlib` module. This helps in constructing paths relative to the project's root directory.
2. **Security Settings**:
   - **`SECRET_KEY`**: A secret key used for cryptographic signing. It should be kept confidential and changed for production use.
   - **`DEBUG`**: A Boolean indicating whether the app is in debug mode. It should be set to `False` in production for security.
   - **`ALLOWED_HOSTS`**: A list of strings representing the host/domain names that the Django site can serve.
3. **CSRF and CORS Settings**:
   - **`CSRF_COOKIE_DOMAIN`**: Controls which domain the CSRF cookie is sent to.
   - **`CORS_ALLOW_ALL_ORIGINS`** and **`CORS_ORIGIN_ALLOW_ALL`**: Allow cross-origin requests. This is generally for development only and should be configured more securely in production.
4. **Installed Applications**:
   - **`INSTALLED_APPS`**: Lists all Django and third-party apps that are installed and activated in the project. This includes core Django apps like `django.contrib.admin`, as well as custom apps like `textanalysis` and third-party packages like `corsheaders`.
5. **Middleware**:
   - **`MIDDLEWARE`**: A list of middleware classes that process requests and responses. This includes security middleware, session management, CSRF protection, authentication middleware, and `corsheaders.middleware.CorsMiddleware` for handling cross-origin requests.
6. **URL Configuration**:
   - **`ROOT_URLCONF`**: Specifies the module that contains the URL configurations for the project (`simplyclip_backend.urls`).
7. **Template Settings**:
   - **`TEMPLATES`**: Configuration for the Django template system. It specifies template directories and context processors used during template rendering.

8. **WSGI Configuration**:
   - **`WSGI_APPLICATION`**: Specifies the WSGI application callable, which serves as the entry point for WSGI-compatible web servers to serve the Django project.
9. **Database Configuration**:
   - **`DATABASES`**: Defines the database setup. In this case, it uses SQLite as the database engine (`django.db.backends.sqlite3`) and specifies the database file path.
10. **Password Validation**:
   - **`AUTH_PASSWORD_VALIDATORS`**: A list of password validation rules to ensure strong passwords for user accounts.
11. **Internationalization**:
   - **`LANGUAGE_CODE`**: Specifies the default language code for the project.
   - **`TIME_ZONE`**: Sets the project's time zone.
   - **`USE_I18N`**, **`USE_L10N`**, and **`USE_TZ`**: Configures settings for internationalization, localization, and time zone support.
12. **Static Files**:
   - **`STATIC_URL`**: The URL path for serving static files (e.g., CSS, JavaScript).
13. **Primary Key Field Type**:
   - **`DEFAULT_AUTO_FIELD`**: Specifies the type of primary key field used for models by default (`django.db.models.BigAutoField`).

MANIFEST.JSON

The `manifest.json` file is a crucial part of any Chrome extension, serving as the configuration file that defines the extension's metadata, permissions, scripts, and overall behavior. It is used by the Chrome browser to understand how to integrate and operate the extension.

Key Components and Their Purposes

1. **`name`**:
   - **Description**: Specifies the name of the Chrome extension.
   - **Value**: `"SimplyClip"`
   - **Purpose**: This is the name users will see in the Chrome Web Store and browser extensions page.
2. **`description`**:
   - **Description**: Provides a brief description of what the extension does.
   - **Value**: `"A clipboard manager for chrome based browsers"`
   - **Purpose**: Helps users understand the extension's functionality at a glance.
3. **`manifest_version`**:
   - **Description**: Specifies the version of the manifest file format.
   - **Value**: 2

    - **Purpose**: Indicates that the extension uses Manifest Version 2, which outlines the supported capabilities and structure for Chrome extensions.
4. **`version`**:
    - **Description**: Indicates the version of the extension.
    - **Value**: `"5.0"`
    - **Purpose**: Useful for version control and updates, allowing developers and users to track the version of the extension.
5. **`content_scripts`**:
    - **Description**: Defines scripts that run in the context of web pages.
    - **Details**:
        - **`matches`**: Specifies the URLs or patterns that the scripts should match. `"<all_urls>"` means the scripts will run on all pages.
        - **`js`**: Lists the JavaScript files that are injected into the pages (`"content.js"` and `"./test/test.js"`).
    - **Purpose**: Enables the extension to inject and run scripts on webpages for tasks like reading clipboard data or interacting with page content.
6. **`background`**:
    - **Description**: Specifies the script that runs in the background, continuously or in response to events.
    - **Value**: `"scripts": ["background.js"]`
    - **Purpose**: The background script manages long-running tasks and event handling, such as context menu actions or monitoring clipboard changes.
7. **`icons`**:
    - **Description**: Provides icons for the extension at different sizes.
    - **Details**:
        - **32**: Refers to a 32x32 pixel icon (`"./images/paper-note-32.png"`).
    - **Purpose**: Represents the extension visually in the toolbar and Chrome Web Store.
8. **`options_page`**:
    - **Description**: Specifies the HTML file for the options or settings page.
    - **Value**: `"./test/test.html"`
    - **Purpose**: Allows users to customize or configure the extension's behavior.
9. **`browser_action`**:
    - **Description**: Defines the UI element shown in the browser's toolbar.
    - **Details**:
        - **`default_icon`**: The icon displayed in the browser toolbar.
        - **`default_title`**: The tooltip text shown when hovering over the extension icon.
        - **`default_popup`**: The HTML file (`"popup.html"`) displayed when the extension icon is clicked.
    - **Purpose**: Provides an interactive UI for users to engage with the extension, showing a popup with options or data.

10. **`permissions`**:
    - **Description**: Lists the permissions required by the extension to function.
    - **Details**:
        - **`"storage"`**: Allows the extension to store data using Chrome's storage API.
        - **`"tabs"`**: Grants access to interact with browser tabs.
        - **`"clipboardRead"` and `"clipboardWrite"`**: Permissions for reading and writing clipboard data.
        - **`"activeTab"`**: Allows the extension to interact with the currently active tab when invoked.
        - **`"contextMenus"`**: Enables the use of custom context menu items.
        - **`"http://127.0.0.1/*"`**: Grants access to local server resources for testing or interactions with a local backend.
    - **Purpose**: Ensures the extension has the necessary permissions to operate fully, such as reading from and writing to the clipboard, and managing browser tabs.