Testing Plan

For now, the only testing that can be effectively completed must be done through the backend. All these test plans deal with data handling and validation to ensure the program is behaving as expected and we are not surprised if abnormal data enters any of our methods.

I realize things are a bit all over the place, but this will hopefully give a good overview of the backend features that will be implemented throughout development. Already from this list, features relating to querying and storage should be prioritized over features like sorting and transferring (as they serve as the basis for additional features).

All testing plans assume that type checking will occur (i.e. passing in None does not crash the application).

Querying (High Priority)

- Case 1: Retrieving all currently hosted ZOIA patches from the PS API (Very high priority)
    - The query must ensure that **ALL ZOIA** patches currently hosted are retrieved. The test should compare the number of patches listed on the PS website to the number of patch objects retrieved and processed.
    - The query should ensure that only the necessary data is retrieved from PatchStorage. Based on the BaseSchema, this includes the following:
        - The patch id
        - The created_at attribute
        - The patch title
        - The content attribute
        - The files attribute
        - The preview_url attribute
        - The view_count attribute
        - The like_count attribute
        - The download_count attribute
        - The patch author (an item containing an id and name)
        - The categories for the patch (an array of category items containing an id and name)
        - The tags for the patch (an array of tag items containing an id and name)
        - The state of the patch
        - The license of the patch
        - The custom_license_text attribute
    - The test should ensure additional unnecessary data is **not** returned by the query to ensure that the querying process is as efficient and meaningful as possible.
    - The method must inform another backend method to begin checking previously downloaded patches for updates. The method must conclude by returning a list of JSON objects that contains the above data for all ZOIA patches retrieved from PatchStorage, which will be used by the frontend for display purposes.

- Case 2: Retrieving a binary file from the PS API (High priority)
    - The method will require a JSON object to function. This object will be passed from the frontend. It must contain an id, url, filesize, and filename. The method should ensure that it functions correctly if the filesize or filename attributes are missing.
    - The method should gracefully handle situations where the patch could not be downloaded and promptly inform the appropriate party in such a situation.
    - The query should retrieve the file and should perform a subsequent query to retrieve the revision, self, and link attributes for the patch that is being downloaded.
    - The binary file that is retrieved should be compared against the filesize and filename attributes of the passed JSON object if they exist.
    - The method must identify the file extension as .bin save the binary file to the appropriate local directory, inform a backend method that new metadata must be prepared, and inform the frontend upon successful completion of this operation.
- Case 3: Retrieving a compressed file from the PS API (High priority)
    - The method will require a JSON object to function. This object will be passed from the frontend. It must contain an id, url, filesize, and filename. The method should ensure that it functions correctly if the filesize or filename attributes are missing.
    - The method should gracefully handle situations where the patch could not be downloaded and promptly inform the appropriate party in such a situation.
    - The query should retrieve the file and should perform a subsequent query to retrieve the revision, self, and link attributes for the patch that is being downloaded.
    - The zip file that is retrieved should be compared against the filesize and filename attributes of the passed JSON object if they exist.
    - The method must identify the file extension as a compressed file. It should be uncompressed, and the method must analyze the contents, looking for files with the .bin extension. Any .txt files should be scrapped and stored temporarily.
    - For every .bin file found, they should be saved to the appropriate local directory. Once this is done, a backend method should be informed that new metadata must be created for each .bin file identified; passing the previously-stored .txt file contents if applicable to be appended to the content attribute for each patch. The method must conclude by informing the appropriate party upon the successful completion of all operations.
- Case 4: Retrieving an updated file from the PS API. (Complicated – Long term)
    - Upon start-up, a check must be performed after retrieval of all ZOIA patches currently stored on PatchStorage. This will require a special PatchStorage API query that retrieves the revision, files, and id attributes for patches stored on the local filesystem.
    - The test should ensure no additional information is returned by the above query.
    - The method must parse all metadata stored on the local filesystem against the metadata retrieved from PatchStorage query. If the revision or filesize differs, the method should retrieve the file from the PatchStorage API.
    - If this retrieved file is different from compared to the locally stored version, a new directory must be created to contain both the previous and new versions of a patch (if such a directory did not exist in the past).
    - If the method cannot retrieve a file, it should gracefully inform an appropriate party that it failed to perform its function.
    - Upon completion, the method should inform the appropriate party that the retrieval of updated patches completed successfully.

Sorting (Low Priority)

- All sorting cases will require that the sorting implements the best Big O sorting algorithm to ensure that it concludes in a reasonable amount of time. Cases that deal with integer sorting should ideally implement radix or counting sorting algorithms.
- Case 1: Sort by patch title. (Very low priority)
    - Sorting must be done by checking the title of each patch and performing a sort in lexicographical or reverse lexicographical order. This ordering depends on the specified order boolean parameter.
    - As specified by the BaseSchema, titles are required, however, the method should be able to handle cases where a title does not exist without crashing.
    - The sorting method must conclude by returning a list of sorted patch metadata (sorted in correct lexicographical order) to be used by the frontend for display purposes.
    - This sorting can occur both from using the locally stored metadata for locally stored patches or the temporary metadata as retrieved via the PatchStorage API. Depending on the context, the method should either access the local filesystem or sort the temporary metadata that was retrieved as the application began.
- Case 2: Sort by tags. (Very low priority)
    - Sorting must be done by checking the tags associated with each patch and performing a sort in lexicographical or reverse lexicographical order. This ordering depends on the specified order boolean parameter.
    - As a patch can have multiple tags, the exact specification for the sorting is not yet known. This will be updated once a decision has been reached.
    - As specified by the BaseSchema, tags are **not** required. The method should be able to handle cases where a patch does not have tags without crashing. (**Suggestions for how to add these untagged patches to the sorted list would be appreciated**)
    - The sorting method must conclude by returning a list of sorted patch metadata (sorted in correct lexicographical order) to be used by the frontend for display purposes.
    - This sorting can occur both from using the locally stored metadata or the temporary metadata as retrieved via the PatchStorage API. Depending on the context, the method should either access the local filesystem or sort the temporary metadata that was retrieved as the application began.
- Case 3: Sort by categories. (Very low priority)
    - Sorting must be done by checking the categories associated with each patch and performing a sort in lexicographical or reverse lexicographical order. This ordering depends on the specified order boolean parameter.
    - As a patch can have multiple categories, the exact specification for the sorting is not yet known. This will be updated once a decision has been reached.
    - As specified by the BaseSchema, categories are **not** required. The method should be able to handle cases where a patch does not have categories without crashing. (**Suggestions for how to add these uncategorized patches to the sorted list would be appreciated**)
    - The sorting method must conclude by returning a list of sorted patch metadata (sorted in correct lexicographical order) to be used by the frontend for display purposes.
    - This sorting can occur both from using the locally stored metadata or the temporary metadata as retrieved via the PatchStorage API. Depending on the

context, the method should either access the local filesystem or sort the temporary metadata that was retrieved as the application began.

- Case 4: Sort by author. (Very low priority)
    - Sorting must be done by checking the author's name associated with each patch and performing a sort in lexicographical or reverse lexicographical order. This ordering depends on the specified order boolean parameter.
    - As specified by the BaseSchema, an author's name is **not** required. The method should be able to handle cases where a patch does not have an author without crashing. (**Suggestions for how to add these unauthored patches to the sorted list would be appreciated**)
    - The sorting method must conclude by returning a list of sorted patch metadata (sorted in correct lexicographical order) to be used by the frontend for display purposes.
    - This sorting can occur both from using the locally stored metadata or the temporary metadata as retrieved via the PatchStorage API. Depending on the context, the method should either access the local filesystem or sort the temporary metadata that was retrieved as the application began.
- Case 5: Sort by date modified. (Very low priority)
    - Sorting must be done by checking the date modified associated with the metadata file itself (not contained within the json) and performing a sort in lexicographical or reverse lexicographical order. This ordering depends on the specified order boolean parameter.
    - All files contain a date modified as apart of the file metadata. However, the method should be able to process files where this date is hidden or inaccessible. (**Suggestions for how to add these types of files to the sorted list would be appreciated**)
    - The sorting method must conclude by returning a list of sorted patch metadata (sorted in correct lexicographical order) to be used by the frontend for display purposes.
    - This sorting can occur both from using the locally stored metadata or the temporary metadata as retrieved via the PatchStorage API. Depending on the context, the method should either access the local filesystem or sort the temporary metadata that was retrieved as the application began.

Naming and Importing (High priority)

- Case 1: Files are added to the LibraryApp directory that originate from PatchStorage.
    - The name of the newly added file should reflect the patch id stored within the metadata of the patch that was just retrieved.
    - The method should gracefully handle the cases where the file could not be named the patch id (either lack of permission, naming conflict, or something else).
    - The method should inform the appropriate party upon completion of the naming.
- Case 2: Files are imported into the LibraryApp via SD or a different local location and they did not exist previously in the LibraryApp directory.
    - The method should assign a unique **5-digit** identifier that will be used as the patch id.
    - Metadata must be created for each imported patch, comprising of the required attribute of patch id, title, and created_at.

    - The name of these patches and accompanying metadata should be updated to conform to the newly assigned 5-digit identifier.
    - The method should gracefully handle the cases where the file could not be named the patch id (either lack of permission, naming conflict, or something else).
    - The method should inform the appropriate party upon completion of importing and naming.
- Case 3: Files are imported into the LibraryApp via SD or a different local location and they did exist previously in the LibraryApp directory.
    - The method should determine if the patch that will be important is identical to a patch already stored in the LibraryApp folder (comparing the byte data of the binary files).
    - Should the file already exist, the method should inform the appropriate party that the file already exists.
    - Should the file be named the same as another currently stored patch (by checking the title attribute in all patches metadata), the method should inform the appropriate party that such a situation has occurred.
    - Should a user insist this is a new patch, the procedure must match the procedure presented in Case 2.

Retrieving (Locally stored patches specifically, not via the PS API) (Medium Priority)

- The method will require a file path to begin. This file path should be validated to ensure the path is in the correct format.
- The method should ensure that the file path points to an actual object. If the file path is deemed invalid, the method should gracefully handle the situation and inform the appropriate party.
- Upon completion, the method must return the file at the passed file path.

Directory Creation (High Priority)

- Case 1: The application is launched.
    - The method should assume that the application has never been launched.
    - The method should successfully locate the appropriate local file location (AppData, Library, etc.) and create a LibraryApp directory (if it does not already exist) based on the determine OS.
    - The method should make every non-invasive effort to make the LibraryApp directory a hidden directory.
    - The test should ensure the directory was created in the expected location based on the determined OS.
    - The method should inform the appropriate party upon the successful creation of the LibraryApp directory.
    - On subsequent start-ups, the method should ensure that the LibraryApp directory exists and inform the appropriate party of this fact. If it does not, it should create the directory as outlined above.

- Case 2: Multiple versions of a patch exist.
    - The method should create a patch directory using the patch id of the patch that now has multiple versions within the LibraryApp directory.
    - The method should move the versions of the patch and accompanying metadata to this newly created directory.
    - If the directory could not be created or the patches could not be moved, the method should gracefully inform the appropriate party of this result.
    - The test should ensure that the directory is created successfully, and the files are located within the newly created directory.
    - The method must inform the appropriate party upon the successful completion of the operation.

Metadata Creation (High Priority)

- Case 1: Metadata does not exist for a patch.
    - The method should be passed the required information that will be used to construct the metadata file.
    - The method will prepare a metadata file, with the necessary attributes, that will be named using the id attribute.
    - The metadata file will be stored within the LibraryApp directory.
    - Should the metadata fail to create, or the file cannot be stored within the LibraryApp directory, the method must gracefully handle the situation and inform the appropriate party.
    - The test should ensure that the metadata was created correctly, and all passed attributes are present and stored correctly.
    - The method must inform the appropriate party upon the successful creation of the metadata for a patch.
- Case 2: Metadata already exists for a patch
    - The method should be passed the required information that will be used to construct the metadata file.
    - The method should correctly identify that metadata for the passed patch id already exists.
    - The method should update the fields within the metadata that differ from the passed required information.
    - The test should ensure the updating of metadata was done correctly.
    - The method must inform the appropriate party upon the successful creation of the metadata for a patch.

Deletion (Low Priority)

- Case 1: Deleting a patch that is stored on the local filesystem.
    - The method should ensure that the patch exists in the local directory. If it cannot be found, it should gracefully inform the appropriate party of this fact.
    - The method should ensure that the desired file **and** accompanying metadata file is deleted.
    - The test should ensure only the specified file and accompanying metadata files are deleted. If the deletion results in an empty version directory, this directory should also be deleted, and the test should check to ensure it no longer exists.
    - Upon successful completion, the method should inform the appropriate party that the deletion has completed.
- Case 2: Deleting a patch that is stored on an SD card.
    - The method should ensure that the patch exists in the local directory. If it cannot be found, it should gracefully inform the appropriate party of this fact.
    - The method should ensure that the desired file is deleted.
    - The test should ensure only the specified file is deleted.
    - Upon successful completion, the method should inform the appropriate party that the deletion has completed.

SD Card (Medium Priority)

- Case 1: No SD card is inserted
    - The method should inform the appropriate party if no SD card is detected.
    - The method should also provide information that aids to troubleshoot common problems that may hinder the detection of SD cards that are inserted.
- Case 2: One or more SD cards are inserted
    - The method should collect the location and names of all inserted SD cards.
    - The method should ensure that the contents of each inserted SD card match the format expected such that it can accurately interface with a ZOIA.
    - Should an SD card lack the appropriate file structure, the method should inform the appropriate party of such a situation.
    - The method must return a list of all SD cards that are properly formatted.
- Case 3: Transferring to an SD card
    - The method must be supplied with a file path that points to the location of the patch on the local system. The file path should be validated to ensure it is in the correct format before proceeding.
    - The method must gracefully handle situations where the file path does not point to an object and must inform the appropriate party of such situations.
    - The method must retrieve the file and rename it to conform with the title attribute stored in that patches metadata.
    - The method must add an appropriate number prefix to ensure that it conforms to the SD card slot it is being transferred into.
    - The test should validate that the prefix matches the slot and that the name is the same as the title attribute if it exists.
    - The method must inform the appropriate party upon completion of this transfer.

- Case 4: Transferring off an SD card and the patch is already exists on the local filesystem
    - The method must be supplied with a file path that points to the location of the patch on the local system. The file path should be validated to ensure it is in the correct format before proceeding.
    - The method must gracefully handle situations where the file path does not point to an object and must inform the appropriate party of such situations.
    - The method must retrieve the file and rename it to conform with the title attribute stored in that patches metadata.
    - The method must add an appropriate number prefix to ensure that it conforms to the SD card slot it is being transferred into.
    - The test should validate that the prefix matches the slot and that the name is the same as the title attribute if it exists.
    - The method must inform the appropriate party upon completion of this transfer.
- Case 5: Transferring off an SD card and the patch does not exist on the local filesystem
    - The method must be supplied with a file path that points to the location of the patch on the local system. The file path should be validated to ensure it is in the correct format before proceeding.
    - The method must gracefully handle situations where the file path does not point to an object and must inform the appropriate party of such situations.
    - The method must retrieve the file and rename it to conform with the title attribute stored in that patches metadata.
    - The method must add an appropriate number prefix to ensure that it conforms to the SD card slot it is being transferred into.
    - The test should validate that the prefix matches the slot and that the name is the same as the title attribute if it exists.
    - The method must inform the appropriate party upon completion of this transfer.

Banks (Low Priority)

- These require a lot of thought. I do not think we can guess at use cases or come up with meaningful test cases until we figure out the implementation of other backend methods first.