**Implementation**

**Why we chose Electron**

The desktop application was developed on the Electron JS platform. Electron uses Chromium and NodeJS, so the application was built using HTML, CSS, and JavaScript. Electron offers great cross platform compatibility and with a single code base for all the major platforms, the packaging and distribution to Mac, Windows, and Linux, is easy. User Interface (UI) and User Experience (UX) can be the same to all platforms and are easy to handle since it is mainly web design. It provides extended interactive features such as keyboard shortcuts and low-level accessibility to the hardware and operations system components. Electron separates clearly the components by making use of separate processes for each web page and a separate, main process that is used to interact with any backend/server systems.

When opening the application, the login form is shown to the user. After the successful login, the user is prompted to select the folder that will be checked and synchronized with the server application. With the folder selection, the process of reading the local files within the folder starts. First, we iterate through the local files through the folder chosen and we get some statistics like name, size and their hash and we save all the information in an array of json objects for easy access. Following this, a GET request - containing the user’s cookie as received from the login - to the server is sent to retrieve the logged in user’s files that are uploaded to the server. When the response is received, the received information is compared against the local files array in order to check their status. The server response contains a hash of the content and the file version that is uploaded on the server. Each file is represented as an object with attributes that include: flags showing if the file is synchronized and whether the content is different, and the version number. These flags make both the functionality and the visualization easier as based on them, the appropriate options and icons can be given to the user.

Subsequently, the files list is rendered on the screen. For all the files the user has the option of deleting them either locally, or from the server or from both sides. For the files that exist only locally, the user has the option of uploading them to the server, and likewise, if the files exist only on the server, the user has the option to download them in the folder specified in the application start. In case the files exist both locally and on the server, there are multiple cases:

* + The files have the same version and the same hash content: in this case the file is up to date and the user does not have any options available except for opening the file with the specified default program for opening the files of this filetype in the OS
  + The files have the same version but different hash contents: in this case the file was changed by the user locally, so he gets the option to upload the file to the server
  + The files have different versions: in this case, someone else has modified and uploaded the file on the server, and the user has a different version of the file. As a result, he gets the options of forcing uploading or downloading and overwriting the contents of either the server of the local file. In case of a simple text file, the option to see the file differences in a new window is given to the user

The option to logout is accessible from every screen. Upon logging out all local storage kept for the application is deleted. *This was decided mainly as the process of dealing with multiple users wanting to synchronize files from the same folder would require a different approach to the way we save files in the server and the way we store information about files’ status locally.* Finally, the user has the option to delete his account.

**Libraries**

The request library was used to handle all the API requests to the server as it offers cookie handling, response streaming to any file type as well as supporting both promises and callback interface natively. In order to use the request library easier, the util module from NodeJS was also used in conjunction to it, in order to handle the requests as promises and not dealing with callbacks.

The fs module of NodeJS was used to interact with the file system as we needed to handle files (open/close/delete/write) and read statistics about them.

The crypto module of NodeJS was used to create the local files’ sha256 hash in order to use it to compare it with what the server sends back so we can check quickly for differences

The electron json storage library was used to persist and read user settings for the application. Upon logging in, the selected folder was saved as a setting so as the user wouldn’t have to select the folder again – unless he logged out.

The isbinaryfile module was used to check if a file is binary or simply a text file so that we check if we can show the file differences simply in the app or not.

Diff2html was used to visualize the differences between the local and server files in case of simple text files.

[1] <https://github.com/request/request>

[2] <https://nodejs.org/docs/v0.3.1/api/fs.html>

[3] <https://nodejs.org/api/crypto.html>

[4] <https://www.npmjs.com/package/electron-json-storage>

[5] <https://www.npmjs.com/package/isbinaryfile>

[6] <https://diff2html.xyz/>