# Cross Platform Compatibility in Open Source Calendar Applications


UNIVERSITY OF
LINCOLN

Joshua Parker
**25197848**

25197848@students.lincoln.ac.uk
joshtsparkee@gmail.com

School of Computer Science
College of Science
University of Lincoln

Submitted in partial fulfilment of the requirements for the Degree of Computer Science

Supervisor: Charles Fox

May 2023

## Abstract

This project aims to improve the Radicale open source calendar application by providing built in integration with various other calendar applications. Specifically this project will allow users of Google Calendar, and Outlook Calendar to easily import their calendars into Radicale, either copying them and allowing them to read and write to the calendar, or keeping them read only, but allowing auto-updating if the original calendar is edited. Keeping all their calendars stored in a single place, would allow a more easy overview of all a user's calendars, and make the move from other calendar services to Radicale much more simple.

## Table of Contents

# 1 Introduction

## 1.1 Motivation for project

The motivation for this project is the desire to contribute to an open source program, wishing to implement an improvement to a program already in use and aid those who use it. Radicale was chosen for a few reasons, firstly it was already a useful tool with many features that could be built upon. Secondly, due to FreedomBox choosing it as their CalDAV/CardDAV server of choice, it already has an existing user base. Finally, calendars are an extremely important part of day to day life, and with the personal data big companies like Google or Microsoft are collecting and selling, adding functionality to an open source calendar, making it easier for people to move to them from these bigger companies' calendars, is a worthwhile goal.

## 1.2 Terminology

There are a few terms it's important to understand to ensure this report is correctly interpreted. To this end a selection of important terminology is discussed here.

Scenes - When scenes are mentioned in this report, it's referencing different displays of a webpage. For example this program contains a 'login scene' in which the user inputs a username and password, and then leads to a 'collections scene' where all of that user's calendars are displayed. This is distinct from a webpage as all the scenes are on a single page; pressing the back button won't take the user to the previous scene, but to the webpage they were on before opening the Radicale web interface.

Calendars - While this project's primary goal is to allow compatibility between Radicale and other calendar applications (such as Google Calendar), given that Radicale also stores tasks and address books, the created program will also allow users to import those through the same method. When 'calendars' are referred to in this report, it is referring to calendars, address books, and tasks.

Collections - A collection is the format some information of a calendar is stored in Radicale. Each calendar's folder has a .props file which contains its href, type, displayname, description, and colour values. The web interface then reads these values for each calendar, displaying this information about them all, and allowing the users to copy the links which let them access each calendar.

Project vs Program - For the purposes of this report, when 'the program' is mentioned, it's referring to the existing Radicale codebase, and when 'the project' is mentioned (in reference to the code) is the additions to this codebase that are being made.

# 2 Literature Review

## 2.1 Aims and Objectives

### 2.1.1 Aims

At its most simple level, the aims for this project are threefold, firstly, to allow the user to import a calendar Radicale through its web application, secondly, to allow the user customizability in importing a calendar, the colour they want to appear by its collection, whether it should automatically update or not, and finally, if the user wishes their calendar to automatically update, the calendar should update consistently.

### 2.1.2 Objectives

Expanding somewhat on the more aims, for this project to be a success, a few steps must be accomplished. Firstly, an addition must be made to the Radicale web interface, this addition must allow users to input a URL, decide if

they want the imported calendar to be read/write or read only and automatically updating, and similar to general creation of calendars on the interface, users must be able to select a colour to appear next to their calendar.

Secondly, the program should be able to import the calendar stored at the given URL into the Radicale calendar application, displaying errors if the provided URL doesn't link to a calendar.

Thirdly, if the user has asked for their calendar to automatically update, the calendar they've imported must be able to re-import itself from the given URL on a consistent cycle, replacing the contents of the last time it was imported with the current contents of the calendar. This replacement of course must only happen when the URL is readable, ensuring that the calendar is never simply wiped as it can't access the URL.

Finally as this project is an addition to an existing program, the project must not interfere with the pre-existing code; It must all run smoothly alongside each other. The project must also maintain the same structure and formatting as the pre-existing code, for example, as the Radicale code has very few comments, the additions made to the program must also have few comments.

These technical objectives should all be completed at a minimum one week before the project's final deadline (i.e. by the fourth of May) to give ample time to review the code and perform final checks.

## 2.2 Core Literature Review

*Note: Much of this section (section 2.2) has been taken from the interim report for this same project, and then updated.*

Radicale provides an open source, server-side calendar server, which allows the storing of multiple calendars, and in which calendars can be created, uploaded, or downloaded. The web interface displays collections linked to these calendars allowing users to distinctly identify them, these collections properties can also be adjusted through the web interface. It also has many useful features such as the ability to limit access to calendars by authentication, storing all the data on the file system in a simple folder structure, and compatibility with many existing CalDAV clients[1] (such as Firefox's Thunderbird or eMClient). The web interface doesn't directly provide a calendar interface, like what you would find in Google or Apples' calendars, but instead stores the calendar on a server which other calendar applications can read from or edit. If an event is created on a Radicale calendar through Thunderbird application, the event would also appear if one were to view the same calendar through the eMClient calendar interface.

Improving open source calendars is an important task due to the prevalence of digital calendars, the data that companies can draw from them, and the past history of many products that these companies have made. Google, is well known for providing services as a data broker, collecting information on individuals from whatever services they might use, and then selling this information to make money, or using it to show targeted advertisements to users[2], having previously "deliberately impeded and delayed" investigations into their data collection[3]. Additionally, Google is, as most companies are, interested primarily in profits, if a service they've created isn't showing profits anymore, then they will shut it down like they did to Google+ after only eight years[4]. Apple also collects data which it stores unsafely, a continuity protocol previously having led to a privacy leak[5], and Microsoft has, admittedly a long time ago, been investigated by the European Union for violation of their privacy policies[6] though continues to collect similar large amounts of data today[7].

Open source calendar applications, in contrast, cannot read and share your personal information (or if they do, this can be easily checked and removed), and as the code is accessible to all, they also cannot be shut down and can receive updates even if the original creator/s end work on the project (at most they may require another user to host them).

One reason people may not switch from their current, closed source, calendar of choice, is due to work requirements. Many companies will require employees to use a specific calendar application for business, the higher ups and people in charge of scheduling already having experience with that specific application, and keeping all employees on the same system means that timetabling things is extremely easy (this being the system the University of Lincoln's College of Science uses for its employees). Employees in companies that use this system therefore have two options. Firstly, keeping personal events on their work calendars, a system that would allow employers to see their calendars, a privacy issue I'm sure many employees would rather avoid, or secondly to flip between multiple calendar applications to check their availability. This project's specific improvement to the Radicale calendar application will help fix this issue. Through this project, users will be able to easily import calendars from other applications, making their move away from proprietary calendars easier. Additionally, unlike the current implementation of uploading static calendars via Radicale's web interface, calendars imported through this project should be able to be automatically updated, allowing for users to store all their calendars (or at least copies of them) in a single place, never needing to flip back and forth between different places personal and work calendars are stored. Admittedly many calendar applications will allow calendars to be viewed in the same place if given the URL, though the key point of difference is that they'd still be stored in disparate places, not in the same location as this project would allow.

While there are many alternatives to Radicale, such as ownCloud, DAViCal, and Baïkal, Radicale was chosen for this project primarily as it's been chosen by FreedomBox, a free software home server operating system with a specific emphasis on being for non-experts[8], as its calendar server of choice. This already singles it out as a program that works reliably and effectively, and shows that it's a service already in use by people across the globe. Any additional features that could be added to Radicale would therefore assist countless people, and FreedomBox's emphasis on working for non-experts means any improvements to help people move permanently away from proprietary software created by invasive companies are especially useful here.

### 2.3 Technical Overview of Radicale

Given this project is an addition to the Radicale application, it's important to understand exactly how Radicale functions. A brief overview of the various sections is therefore included here as part of this project's literature review.

### 2.3.1 Web Interface Overview

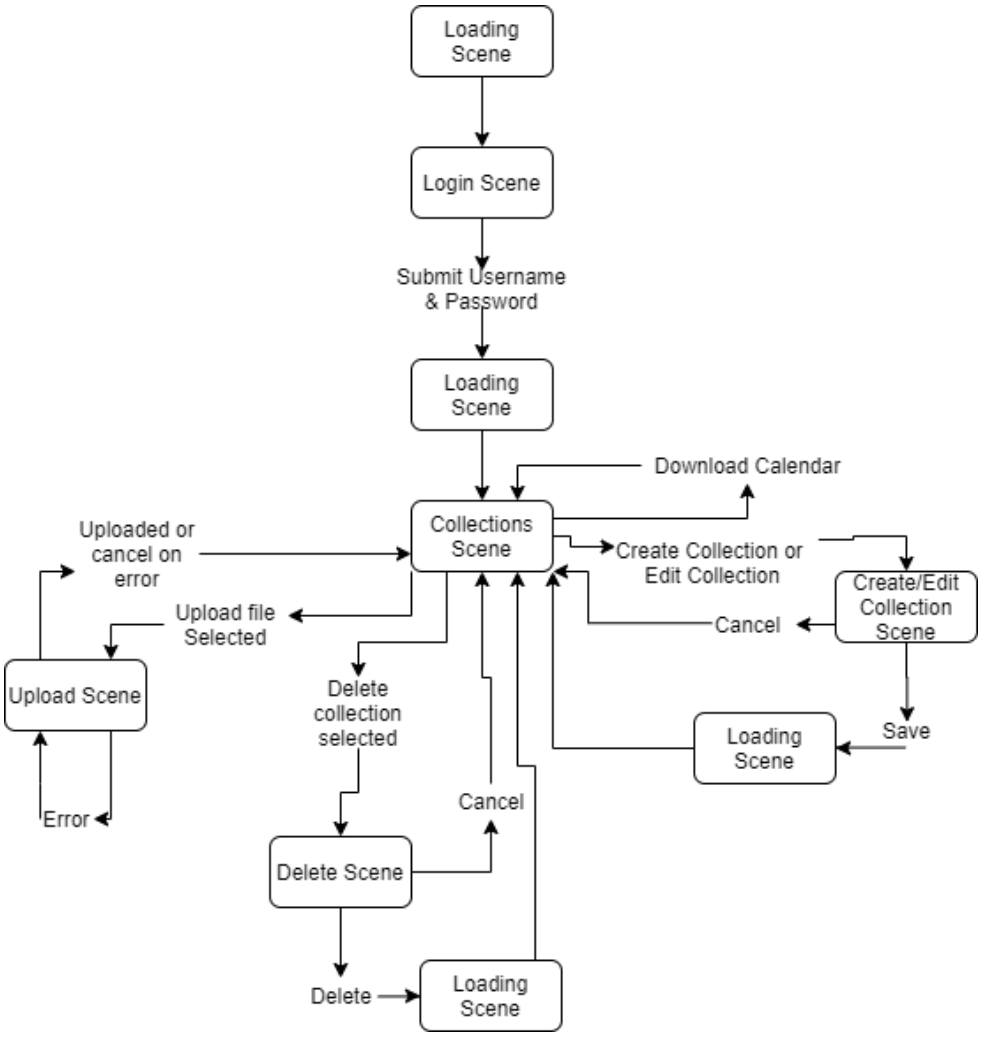



Figure 1 - A Flowchart of scenes in the Radicale web interface

This section will provide an overview of Radicale's web interface, a general flowchart of the various scenes included in which can be seen in figure 1. While this overview is a good simple overview of navigation between scenes, a more detailed explanation of each part of the web interface's functionality will be provided here.

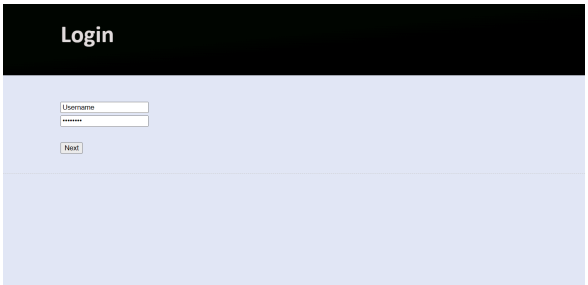



Figure 2 -The Radicale web interface's Login Scene

Ignoring loading screens until the end of this section, whenever a user opens the Radicale web interface they will first be greeted with the login scene. Here a user can input a username and password into two text boxes, the password box not displaying any characters inputted unless the user selects it to, and once they've done these, they can press the 'next' button to move through a loading scene to the collections scene. By default the system doesn't check the given password, however it can easily be adjusted to do so, and is recommended to do so if the server is reachable via a network.

If collections have previously been created when a user was logged in, these created collections will be displayed in the collections scene, information found through the use of a propfind request. If the user hasn't created any collections however, only the two options to do so are displayed: 'Create new addressbook or calendar', and 'Upload addressbook or calendar'.

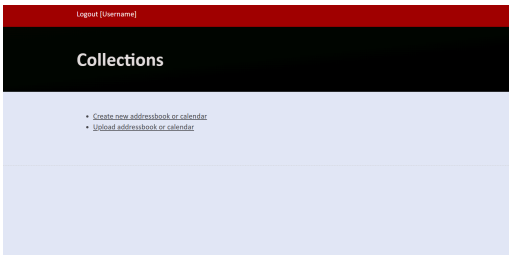


Figure 3 -The Radicale web interface's Collections Scene

Upon selecting the option to upload a calendar, a file input popup will be created, allowing the user to input iCalendar files (or other files of appropriate formats). A user can choose to upload multiple files at once through this method. Once the file/s have been selected, the scene will shift to the 'upload scene', if the user inputs a file containing formatting errors or of an invalid type, they'll be shown an error message and a cancel button that will return them to the collections scene, otherwise they will be returned after the program finishes uploading the given files, making use of a put request. When a calendar is uploaded through this method, its collections' description and colour values are left blank.

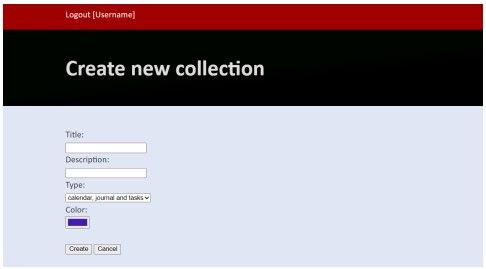


Figure 4 -The Radicale web interface's Create/Edit Collection Scene

The 'Create/Edit Collection Scene' can be accessed through two different ways; Firstly, the user can select the option to create a new addressbook or calendar, and secondly the user can select the option to edit a calendar they've already created or uploaded. If they're editing a calendar that's already been uploaded, the title, description, type, and colour fields will be filled in with those of that collection, however if they're creating a new one, the title and description will be blank, the type field will default to 'calendar, journal and tasks', and the colour will be a randomly generated hex code. In either case, the user can select the cancel button to return to the collections scene, or submit to either create a new collection or adjust the one that's been given. These are done through use of a proppatch request if the program is adjusting a collection, or a mkcol request if the program is creating a new one.

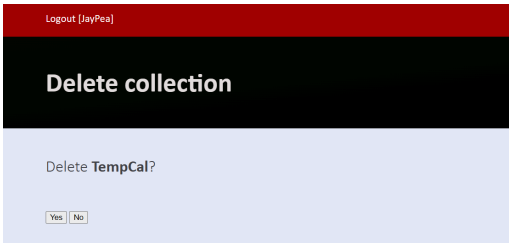


Figure 5 -The Radicale web interface's Delete Scene

Another option a user has to interact with the collections uploaded to the program is to delete them, selecting the delete option for a collection will take the user to a screen where they're asked to confirm that they wish to delete the collection. If they select the cancel button they are returned to the collection scene, but if they select delete, the program makes a delete request and permanently deletes the collection and the information it contains.

Users are also able to download the calendars they upload to the program. The URL for each calendar is, instead of being a link to another scene, linked to the URL for the collection itself, thus selecting it allows the user to download the calendar as an iCalendar file.

When the program is performing a task rather than simply moving between scenes, it will display a loading scene. This is done so that the program has time to finish a process before allowing the user to start another, thus reducing the possibility for errors. For example when the user has chosen to delete a collection, while the collection is being deleted, the program will display a loading scene so the user can't make another request as the first is still ongoing. Additionally if the user had just uploaded a new collection and they were returned to the collection scene before the upload had finished, the newly uploaded collection wouldn't appear on the collection scene, causing potential confusion for the user.

Finally, at any time, no matter the current scene, the user has the option to log out, returning to the login scene. This option isn't included in figure 1 as it's able to occur regardless of the current scene, and overwrites any processes occurring. A user can even log out in the middle of a loading scene, though this may cause errors if the request doesn't finish before the user logs in again.

### 2.3.2 Calendar Storage Overview

This section will discuss how the calendars within the Radicale program are stored. As part of the Radicale program, a storage directory named 'radicale' is created, assuming the program is running on a computer, alongside the user's pictures and documents directories. Within this folder is then stored the calendars that the user has saved within the Radicale program.

Directly within this folder is another folder titled 'collections', within which is another folder titled 'collection-root' and a file, '.Radicale.lock'. This lock file appears empty, and is by default, however, being a .lock file it's in place more for the operating system than the user, many OS' use these files to lock a resource like the collection root folder, but the file itself exists simply as an empty marker to remind the OS, rather than do do anything itself[9]. By default the program doesn't check passwords, though this functionality can be easily set up.

Inside 'collection-root', are more folders, one for each of the different users that have logged into the Radicale web interface, the folder titles being the same as usernames that have been used on the interface. Each of these 'user folders' then contains a folder for each calendar the user has uploaded, the folder names being the same as the href values of each calendar. If the user has created any calendars, the user folder will also each contain a file titled '.Radicale.props', the .props file extension typically denoting a property sheet created by Visual Studio. These .props files contain only a single set of curly brackets, showing that they're likely placeholders that can be called when the program is searching for the calendars; if the program finds this mostly empty .props file, it knows that it's in the right place to start loading calendars.

On the subject of the calendars being loaded, the final layer of files is contained within these calendar folders. These folders first contain another '.Radicale.props' file, however here they aren't blank, instead storing information about the calendar's collection, alongside other information, each calendar's description, display name, and colour is stored here in a standard dictionary format. When the program makes propfind requests, it reads the properties of each collection from these .props files to display them on the interface. Alongside this .props file is a number of iCalendar files, these each contain the different events that make up the calendar itself, when a calendar is imported from Radicale into an application like Thunderbird, this is where the calendar's events are being pulled from.

### 2.3.3 Program Files Overview

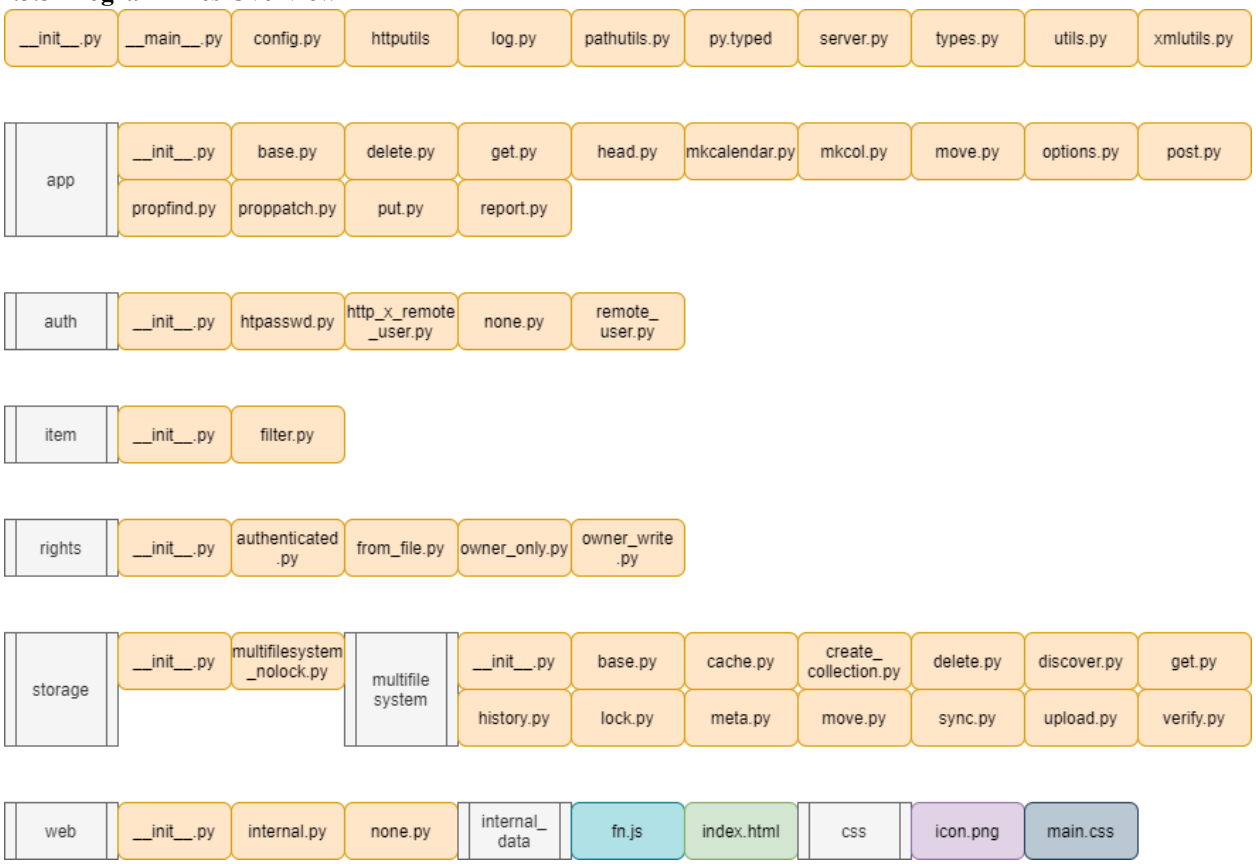



Figure 6 - An overview of files in the Radicale codebase

The final part of the Radicale program that requires an overview is the program files themselves, this section discusses these files, briefly discussing the purpose of each, going through them section by section. Figure 6 shows an overview of all the files included in the program, the white boxes being folders, containing the files (and in some cases folders) to their right. In some cases file and folder names have been split across multiple lines however it should be noted that this was only done for legibility of figure 6, and that the files themselves contain no spaces in their filenames.

The first section of files to discuss is those that are stored at the base directory, not within any other files. '__main__.py' is the most important of all these as this is the file that runs first when the Radicale program is started, this it imports many other parts of the codebase (config, log, server, storage, etc), calling functions from the relevant files (e.g. the serve function from'server.py') to start setting up the program. The 'config.py' file is also important, initialising the program's configuration and containing functions to adjust various parts of the program where needed, for example a function converting given values to booleans or ensuring positive integers. The 'httputils.py' file performs functions related to the web interface, for example if the user uploads a file to the web interface, that file will be sent to a function here, 'read_request_body', to return its contents. Next, 'log.py' is used to keep an active log file of events that occur within the system, being called from many of the various files throughout the program. The 'pathutils.py' file provides various utilities to do with file paths, returning the names from paths, stripping paths, etc. It also provides different path functionality depending on the program's platform, running one way on linux and another on windows, these differences ensure it runs smoothly regardless of platform. The 'server.py' file sets up the Radicale server on which the program runs, and which stores the calendars, allowing calendar applications to read from them. The 'types.py' file contains a context manager function, allowing the

program to be precise in allocation and release of resources, it also performs a function allowing for the union of Radicale's item and storage sections which will be discussed later. The 'utils.py' file contains a couple of functions called in the initialization files discussed later, for example 'lead_plugin' which is used in the auth, rights, storage, and web section's initialization functions. Finally, 'xmlutils.py' contains many functions that would be used in either reading or writing to and from xml files, this is extremely important for this program as the files that store the calendars and information about their collections (iCalendar and props files) can be represented in XML form.

The next section of files is those within the app folder, most of which are called by other parts of the program, often the web interface, when it makes certain requests. The 'base.py' file however isn't one of these, this file contains two classes, 'ApplicationBase' which is passed into a few other classes throughout the files in this folder, and 'Access' which is used to check the access rights of an item. The 'delete.py' file does what the name implies, being used when a delete request is made in the web interface and deleting the collection and calendar given to it. The 'get.py' file is used when starting the web interface, redirecting to the web file when the interface is opened. The 'head.py' file only contains a single class and function within, though is rather important as it's what calls the 'do_GET' function from the get file. The 'mkcalendar.py' file is what's enacted when mkcalendar requests are made, able to make a calendar file itself. In a similar vein, 'mkcol.py' is enacted when mkcol requests are made, performed whenever the user creates a new calendar or uploads one from a file, creating a collection for it. The 'move.py' file is able to enact move requests, changing the locations of given files, however it's currently unable to move full collections. Next are 'options.py' and 'post.py', allowing requests of their namesakes, the former returning all the options a user can enact and the latter performing httppost requests. The 'propfind.py' and 'proppatch.py' files are most easily spoken about together, as with other files here they allow for requests that mirror their filenames, in this case these requests deal with collection properties, find returning all the properties, allowing for collections to be displayed on the collection scene or in editing, and patch adjusting given properties, called when a user is editing a given collection. The 'put.py' file is what allows the user to upload files they pass into the web URL, calling functions from 'httputils.py' to read the contents of the given files and then uploading them. Finally 'report.py' allows the user to obtain information about a given resource.

The next section worth discussing is the files within the auth folder. Firstly 'htpasswd.py' manages passwords for the web interface, supporting both plaintext and MD5-APR1 encryption, using the latter as the default due to how insecure the former is. Next 'http_x_remote_user.py' is another part of the authentication process, taking the username when called and used with a reverse proxy to secure the program. The 'none.py' file is used as a dummy backend, accepting any usernames or passwords provided, the default for the program. Finally 'remote_user.py' is similar to 'http_x_remote_user.py' however it's intended to specifically work with external WSGI servers.

Other than the initialization file, the only file within the item folder is 'filter.py'. This file, like a few before it, contains a number of different functions that interact with various items. A couple of examples of this include checking if an item matches different filters, and checking if an item's time range to ensure it falls within a filter.

The files within the rights folder are all different rights backends, the specific one in use being part of the program's configuration. Firstly, 'authenticated.py' allows authenticated users to read and write all calendars regardless of if it's a calendar they 'own' or not. In contrast 'owner_only.py' allows a user to read and write only their calendars, 'owner_write.py' is then the middle ground between these, allowing users to read all calendars regardless of the owner, but only able to write to their own. Finally 'from_file.py' is based on a regex file from the configuration, allowing more specification on who can do what with each calendar.

The first level of the storage folder contains a file and a folder, the file, 'multifilesystem_nolock.py' is the backend for the multi-file system that runs when there's no file-based locking in place. The contents of this folder then are what allows for the multifile system to function. Firstly 'base.py' allows the base for the system, calling other files in the same directory to start the system, 'cache.py' then, as the name implies, creates a cache of all the files included in the system. The 'create_collection.py' file is what creates the props files (and thus collections) for each

calendar within the system. The ‘delete.py’, ‘get.py’, ‘move.py’, and ‘upload.py’ files are all rather simple to explain, being the storage side of the files in application of the same name, where the folders in ‘app’ are called, requests being made to them, these files are then called to enact the backend of the given request. The ‘discover.py’, ‘sync.py’, ‘verify.py’, and ‘lock.py’ files are also easier to explain, their names explaining their functions, discover is used to find any collections, sync to sync collections with updates from the application, verify to confirm the presence of collections and ensure they aren’t invalid or broken, and lock is used if the file system is intended to be locked. This then leaves the final two files, ‘history.py’, which updates and retrieves the history from calendar’s caches, and ‘meta.py’, which deals in the calendar storage’s metadata.

The web folder’s first level contains two files and a folder. This first file ‘internal.py’ is the default backend for the web interface, and the second, ‘none.py’, is a dummy backend that if run shows the user a message informing them that Radicale is functioning. The folder, ‘internal_data’ then contains two more files and another folder. The html file here, ‘index.html’, contains the structure for the Radicale web interface, and the other file, ‘fn.js’ contains its functions, it’s what allows users to make requests or navigate through scenes on the interface. The folder, appropriately titled ‘css’ contains two more files, ‘main.css’ containing the structures formatting, features such as font, background colour, and border size, and ‘icon.png’, a simple png image of the Radicale logo to display as the web interface’s tab icon.

Finally, there are many files throughout the program with the name ‘__init__.py’ these ‘initialization files are used to mark given directories as python packages, with these files inside given directories, files are able to import elements from other files into themselves, running functions from other files or reading variables from them.

# 3 Requirements Analysis

## 3.1 Constraints

As this project is adding functionality to a pre-existing program, it’s constrained by the fact that it must work within the already pre-established boundaries of the program. Firstly, the project must be able to run on devices that are able to run Radicale, even devices that are less computationally powerful, meaning that the project must be as efficient as possible.

Secondly, the project must not rely on any modules that aren’t pre-installed with the base program or aren’t already included in the program. For example, in python files, the project could use the ‘time’ module as it comes pre-installed with python, or the ‘vobject' module as it’s referred to in the program’s python files, however it couldn’t include the ‘SpeechRecognition’ module as it isn’t pre-installed with python, and isn’t in the program files already.

Finally, the project’s formatting must not be changed, this can be split into two sections. Firstly, the structure of the program should be maintained, spelling and comments for example would both fall under this, something talked about more in section 4.3 of this report. Secondly, the structure of the variables within the program must be kept the same, if the project changes an integer to a string in the wrong place, or adds an extra key and value to a dictionary, this could potentially break the program’s functionality completely, and would require a considerable effort to repair.

## 3.2 Requirements

For this program to be a success, a few requirements must be met. While these were previously mentioned in the aims and objectives section, they have here been further expanded upon to create a list of actionable goals.

1. The user must be able to input a string into the program that the program can store. This basis must be established before the rest of the project is worked on to ensure that users are able to input URLs for the project to read.

2. The project must be able to upload the contents of a calendar stored at a URL, from a URL that it provided to it. Again, this is a requirement for the rest of the project to function at all, so must be completed before other steps are done.

3. After these first two steps are completed, the project should then be able to take a URL submitted by the user, and upload the calendar stored at this location, creating a collection from it.

4. The user must be able to input some details about their calendar that are then stored in the calendar's collection, specifically they should be able to select a colour to appear next to the collection, and should be able to select how exactly they want to import the calendar, either automatically updating (though not allowing them to directly edit it), or simply creating a copy of the calendar onto the Radicale server.

5. While importing a calendar the user shouldn't be able to interact with the rest of the web interface, being shown a loading screen. Additionally, after the calendar is imported they should be returned to the collections scene where their newly imported calendar is displayed.

6. The project must be able to search through all the collections on a consistent basis, checking if the calendar has been set to automatically update, this should occur synchronously with the program itself and not interrupt its processes.

7. If the project finds a calendar collection that has been set to automatically update, it should attempt to re import the calendar from the URL, maintaining the same href. If the import is successful, the calendar's old contents should be deleted.

8. This must all function without error, and if an error is caused for some reason (e.g. the user inputting an invalid URL) error details must be displayed without causing the program to stop functioning.

9. The project must not use any libraries or functions that aren't proven to already exist within the Radicale code, or aren't pre-installed with the language they're being used in. Additionally, the structure of any variables (primarily lists and dictionaries) shouldn't be adjusted. This is done to ensure that were the project to be merged into the Radicale repository it wouldn't break the program for any pre-existing users.

10. In a similar vein to requirement 9, the formatting and structure of the Radicale codebase should also be maintained within the project. If the project is intended to work with an existing program, it should be structured in a way that doesn't stand out and feels like a part of the original program rather than some extra code that's been tacked on to add functionality. This is discussed further in section 4.3 of this report.

# 4 Design & Methodology

## 4.1 Methodology Discussion

The design methodology chosen for this project was a waterfall methodology, completing each section before moving onto the next, chosen for simplicity to manage. Additionally waterfall methodology most frequently falls apart where the team's composition changes frequently, or where there aren't clear requirements and stable objectives, as this project is being completed by a single individual, there's no worry about any changes in the team composition, in addition the objectives and requirements have been clearly set out earlier in the report, sections 2.1.2 and 3.2 respectively, and will not be adjusted.

## 4.2 Relevant Code Overview

The first thing done to prepare for this project was ensuring a grasp on the contents of the pre-existing codebase, specifically the parts relevant to this project. To do this, Radicale was downloaded onto a laptop to allow easy access to its files, and Thunderbird was used as the calendar interface of choice to test this project. To test the base

functionality of Radicale, a few simple calendars were set up through the web interface, and the upload function was tested via a calendar downloaded from Google calendar.

To understand how exactly the program functioned rather than understanding its functions, the code directory, previously discussed in section 2.3.3 of this report was looked at with minor adjustments made. Using print functions and alerts in the python and javascript files respectively helped gain a better understanding of when each file was called and what exactly the programs within entailed. While section 2.3.3 of this report gives a general overview of the entire codebase, this following section will zero in on the parts of the program that were of primary importance to the project. These parts can be split into three sections, firstly the core of the program itself, previously discussed as the files that weren't contained inside a folder in the overview, secondly the web interface, the files contained within the web folder, and finally the python functions contained within the app folder.

Of the files in the core program there were two of primary importance to this project. Firstly the '__main__.py' file, the first file to run on the program's startup and what calls other files to begin the program, and the 'httputils.py', which contains a few different functions that are called by the web interface, most notably the 'read_request_body' function, called when the user inputs a file through the web interface, and is used to import the contents of the given file.

The next section of files of importance are those to do with the web interface, specifically the internal data of the web interface, those within the 'internal_data' folder. Once again there are two main files here of note, firstly 'index.html', this file stores the layout for the web interface itself, and secondly 'fn.js', a javascript file that stores the functionality for the web interface, the code that lets buttons and such on the web interface actually function.

Many of the functions that 'fn,js' allows are only possible however thanks to the third section of the codebase, the python functions, for example, as discussed in previous sections of the report, when a file is deleted, it's done so through calling a function from the 'delete.py' file. The main file of note here is 'put.py', the function that creates a calendar if it's already been given the contents for that calendar.

### 4.3 Codebase Structure

The files in the existing codebase are all structured similarly, a structure that must be maintained when creating this project's addition to the program. Firstly, the program makes use of american english for its variables, as an example whenever the word 'colour' is used, it's instead spelt 'color', because of this wherever words with different spellings in american english and british english are used throughout the code of this project, the american english spelling will be used.

The second thing of note about the program was how sparsely commented it is, in the 'fn.js' file for example, while functions have commented explaining the variable types of each import and occasionally the purpose of the function, and variables created within functions are often commented to explain their types, other than those specific cases it's rather rare to see any in-line comments. This precedence should be maintained throughout the project, very sparsely using comments. To ensure that the exact functionality of the project's code is understood however, a notebook will be kept to track what the code does, allowing the code to be understood, while the project keeps a consistent structure with the program code.

### 4.3 Performance Evaluation

To evaluate the performance of the project, a series of tests will be run to ensure that it meets all the requirements set out in section 3.2 of this report.

1. A calendar that already has an event will be imported from a given URL, providing no colour for the collection and importing it in a way that it doesn't automatically update. This will confirm that at the very least the calendar's title has been read from the URL.

2. This calendar will be imported into Mozilla's Thunderbird application to check that its events have been correctly imported into Radicale.

3. Another calendar will be imported into into Radicale that also isn't intended to auto update, this time the colour provided will be #FFFF00 (or 255, 255, 0) to confirm that the colour inputted is what the program is storing. The image manipulation program GIMP will then be used to confirm the collection's colour.

4. A calendar will be imported into Radicale that will attempt to automatically update. This calendar's collection's .props file will then be checked to confirm that the intent for this calendar to automatically update has been saved.

5. A non-URL string, 'TestString', will attempt to be imported through the web interface. This should cause an error message, but for the program to continue functioning.

6. A URL that doesn't have a calendar stored at it, 'https://www.google.com', will be imported through the web interface again expecting an error message but for the program to continue functioning.

7. The system that checks for auto-updating calendars will be temporarily adjusted to output the names of each calendar that are to be automatically updated, confirming that these calendars are being correctly identified for updating.

8. An automatically updating calendar will be imported into Mozilla Thunderbird, an adjustment will be made to the calendar wherever it's being imported from. The calendar will then be updated in Radicale, and Thunderbird will be refreshed to check for the adjustment.

9. During all the previous imports, attention will be kept to ensure that a user wouldn't be able to make any changes as a calendar is being updated.

10.A new calendar will be created on Radicale to confirm that this project hasn't broken that function of the program.

11. A calendar will be uploaded to Radicale from a file for the same reasons outlined in performance evaluation ten.

12. A raspberry pi with Radicale on it will be set up and use this project's code to import a calendar from a URL on it, confirming that this project, like the original program, can function on less powerful devices.

# 5 Implementation

## 5.1 Implementation of Requirement One

In order to create an addition to the Radicale program, allowing the user to directly upload a program from a given URL, the first thing required for the project was functionality to allow the user to input a URL, and for the program to be able to read this on it being submitted, requirements one, two, and three. The first step to do this was the completion of requirement one, creating a form that could be read from on the main collections scene of the web interface to do so. Given this didn't require reading any information from the given URL, simply displaying the given text, this seemed a rather simple and achievable goal. It also made sense to complete this step first due to the fact that, as discussed in the requirements section, this was an essential requirement for the program.

This task was indeed rather simple, using HTML's built in functionality, a text box that requires URLs could be added to the collection scene of the web interface. Then, using javascript an alert function could be called on submission of the text box, this meant that when a URL was entered into it and the user hit


Figure 7 - An alert showing a data a user inputted to the web interface

enter, a popup would display showing the input the user had given.

### 5.2 Implementation of Requirement Two


Figure 8 - The edited 'read_request_body' function

Now that a basic URL input that could take a given URL from the user had been created, the next step was to look at the python code and see how information could then be read from a given URL, attempting to fulfil requirement two (for now assuming that the given URL is always the link to a calendar). A Google calendar was created and its secret address in iCal format copied to test this. In the existing Radicale codebase there is a function named 'read_request_body' within the 'httputils.py' file, mentioned previously in section 4.2 of the report. This function is used to read the contents of a given file when a put request is made, occurring when a downloaded calendar file is being uploaded to the Radicale repository. In order to test how contents of a URL could be read, for testing purposes the line setting the content variable was commented out, and an attempt was made to instead read the contents of the aforementioned Google calendar. The 'read_raw_request_body' function called in the definition of content returned the actual contents of the file, so to upload the Google calendar, the function call in the content definition needed to be replaced with the contents of the calendar. To this end, python's built in 'urlopen' function was used, a function that allows the program to read from a webpage, after setting a variable, 'file', to the url open of the calendar's secret address, defining content could then be copied and adjusted, replacing the 'read_raw_request_body' function with 'file.read()'. This allowed the program to read the contents of the given URL instead of whatever file was intended to be uploaded. After completing these changes this editing of the function worked perfectly, furthermore because a built-in function was being adjusted, the system already had error handling as needed for requirement eight.

### 5.3 Implementation of Requirement Three

To then combine these two disparate parts, the program's 'UploadCollectionScene' and 'upload_collection' functions were adapted. From these functions, which are called when a file is uploaded via the web interface, the new functions 'URLUploadCollectionScene' and 'url_upload_collection' were created. These would then be called when the user entered a URL into the textbox previously used to create a popup displaying the URL. In theory this would create a new loading scene to prevent the user from taking any actions, satisfying requirement five, upload the contents of the URL as it had done with the temporary resting URL, and then, once the file had been uploaded, return to the collections scene. After duplicating and making the slight adjustments, it became clear that as 'put.py' and 'httputils.py', files used in the uploading of calendars, were both designed to take a file input, adjustments must be made to the contents of those files as well ensuring that they work with URLs. For this adjustment a new function within 'httputils.py' called 'read_request_URL' was created, when a URL was passed as input instead of a file, this function would perform similar actions to the adjustments made to 'read_request_body' for requirement two, performing urlopen on the given URL, and returning its content. An adjustment was then made to 'put.py', checking the content type of the object passed to it, if the object type were 'text/calendar', 'read_request_body' would be called, if it were 'text/plain;charset=UTF-8', 'read_request_url' would be called instead. This change allowed users to import calendars from URLs that they could pass into the web application, thus satisfying requirement three.

While this method did satisfy requirement three, the method by which it functioned wasn't the most intuitive, didn't allow users to select the colour for the calendar they were uploading, didn't allow the user to choose if they wanted the calendar to automatically update (requirement four), and didn't allow the calendar to automatically update at all (requirements six and seven), all of which needed to be fixed for the final product.

### 5.4 Implementation of Requirement Four

The issue it made sense to tackle first was the interface for uploading urls, to satisfy requirement four the user should be able to input the calendar collection's colour and how they want to to upload the calendar, for this the method to upload the calendar would need to be more than just a single text box. For this, in the 'index.html' file the text box previously used to input URLs was replaced with a line of text reading 'Import addressbook or calendar from URL'. A new scene was then constructed, 'URLImportCollectionScene', that could be shown when this text was selected. This scene contained five primary elements: a textbox for the user to input their URL into; a drop-down box with options for what what way they wanted to import the calendar, options reading 'Auto Update, Read Only' and 'No-Update, Read/Write'; a colour selector, similar to the one in the scene for creating or editing calendars; and finally two buttons, 'import' to take information from the prior three elements and import the calendar, and 'cancel' to return to the previous scene. The last part of setting this scene up was adjusting the 'fn.js' file, creating a new function that would show this new scene when the line of text on the collections line was selected, making said function be called when it was, creating an 'onclose' function as part of this scene function to return to the collections scene when it was called, and making sure that this 'onclose' function would be called when the 'close' button was selected.

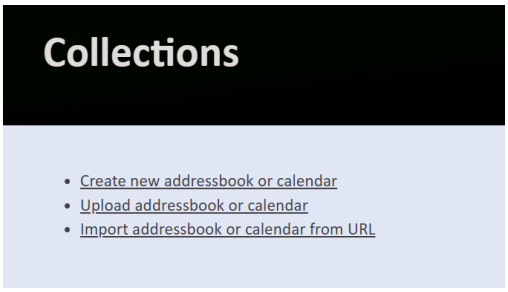



Figure 9 - The updated Collections Scene

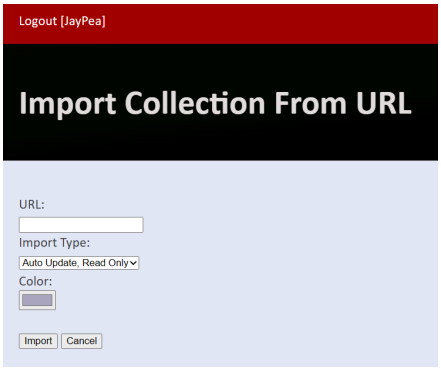



Figure 10 - The newly created Import Collection Scene

While this created the user interface through which users were able to input URLs, the functionality to upload them was still missing. To do this, another new function within the scene was created, one that would be called when the submit button was pressed, this function would, for now, read the URL that the user had put in the text box, and then call the scene that had already been designed to upload given URLs, unfortunately however, calling this scene caused issues when it wasn't being called from the main collections scene.

The intended effect of this code was that the user could go to the import scene from the collections scene, input a URL and select the import button, be taken to the 'URLUploadCollectionScene' while the file was being uploaded, and then be taken back to the collections scene where they could see the new calendar they'd uploaded. The error however was that, by calling the upload scene from a scene other than the collections scene, the callbacks didn't function correctly and the file would still be uploading when the user was returned to the collections scene, thus not showing the newly added calendar until the collections scene was refreshed, and not satisfying requirement five.

The way this error was fixed was simply cutting out 'URLUploadCollectionScene'. While this was needed when being uploaded directly from the collection scene, given the program was now uploading from another scene instead, the code that was being used and called in the aforementioned scene could instead be moved to the function called on the import button's selection in the 'URLImportCollectionScene', creating a loading scene (pre-existing in the codebase) to ensure the user can't perform any inputs while the URL is uploading. Once this code had been moved and adjusted, the scene

```
/**
 * @param {string} user
 * @param {string} password
 * @param {string} collection_href Must always start and end with /.
 * @param {string} url
 * @param {function(?string)} callback Returns error or null
 * @return {XMLHttpRequest}
 */
function url_upload_collection(user, password, collection_href, url, callback) {
    let request = new XMLHttpRequest();
    request.open("PUT", SERVER + collection_href, true, user, password);
    request.onreadystatechange = function () {
        if (request.readyState !== 4) {
            return;
        }
        if (200 <= request.status && request.status < 300) {
            callback(null);
        } else {
            callback(request.status + " " + request.statusText);
        }
    };
    request.setRequestHeader("If-None-Match", "*");
    request.send(url);



    return request;
}
```

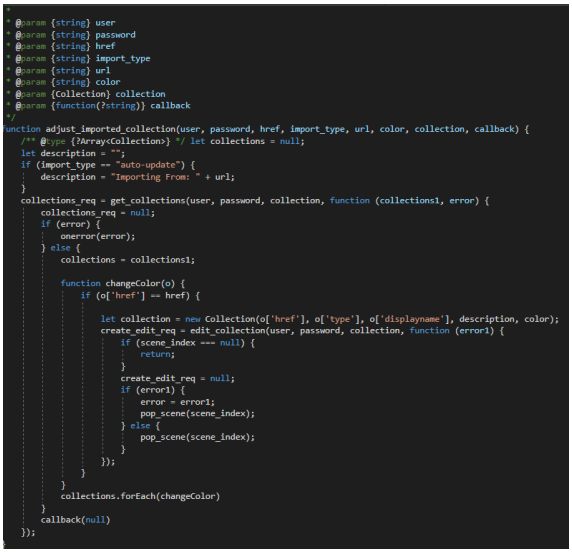
Figure 11 - The function called to upload a file from URL

functioned properly, users able to move easily from the collections scene to the import scene, input a URL and select the import button, be shown a loading scene while the calendar was being uploaded, and then be returned to the collections scene with their newly imported calendar showing, once again satisfying requirement five.

The next task to complete was another part of requirement four, setting the colour of the imported calendar to the colour the user provided when on the import scene. Before doing the main code for this section, a line of code was added to the import scene, so that whenever it was loaded the default value in the colour selector was randomised. After this, a new function was created that could be called to change the colour of the recently uploaded collection. Given the method through which the calendar was uploaded as a collection (using the put function), the program didn't have the uploaded collection to be passed into the function being created so another way to find that collection was needed to then adjust its colour. The first step to doing this was by using the pre-written 'get_collections' function to create a list of all the collections currently stored by the program, after this, the next step was to carefully select what variables would be passed into the function being written. While of course the selected colour would need to be passed in, alongside user, password, and a callback function for error handling to name a few, the most important to be able to find the previously imported calendar was its href value. Because each collection stores its own unique href value, a loop could be performed in which this value of a given collection was checked against the href value of the imported calendar, if they were found to be the same, this given collection's colour value could then be changed to be the same as the colour value that the user imported.

This solution worked, though while the implementation was intended to create a loading scene, run the first function, delete the loading scene then create a new one, run the second function, and finally return to the collection scene, for some reason when uploaded it was getting stuck on a loading screen, despite the colour showing correctly on refreshing the page proving that the function was definitely running correctly. This was quickly fixed however, the issue being that the two functions (uploading and changing colour) were running alongside each other, thus returning the calendar scene before the loading scene was removed, and so never actually removing the second loading scene. The solution was to call the colour changing function from within the upload function's callback ensuring that it must occur after the upload is completed, with this change only one loading scene needed to be created, which could then be removed after the colour changing scene was completed.

After the colour change for the calendar was created, the final part of requirement four was that the project would be able to know which calendars required automatically updating, this being done by storing information about whether the calendar was intended to automatically update somewhere. The ideal solution to do this would be adding another key to each collection object, a boolean that stated if the calendar was an automatically updating calendar or not, or a string storing the URL it was imported from. Unfortunately this change was unfeasible, as discussed in requirement nine adjusting the core structure of the collections could result in previously created or uploaded calendars breaking as their structure doesn't match the new structure for imported calendars, or could break the previously created code as it wouldn't work with the new calendar structure. The solution then settled on instead was to use a part of the collection's structure that was already there.



```
/**
 * @param {string} user
 * @param {string} password
 * @param {string} href
 * @param {string} import_type
 * @param {string} url
 * @param {string} color
 * @param {Collection} collection
 * @param {function(?string)} callback
 */
function adjust_imported_collection(user, password, href, import_type, url, color, collection, callback) {
    /** @type {?Array<Collection>} */ let collections = null;
    let description = "";
    if (import_type == "auto-update") {
        description = "Importing From: " + url;
    }
    collections_req = get_collections(user, password, collection, function (collections1, error) {
        collections_req = null;
        if (error) {
            onerror(error);
        } else {
            collections = collections1;

            function changeColor(o) {
                if (o['href'] == href) {

                    let collection = new Collection(o['href'], o['type'], o['displayname'], description, color);
                    create_edit_req = edit_collection(user, password, collection, function (error1) {
                        if (scene_index === null) {
                            return;
                        }
                        create_edit_req = null;
                        if (error1) {
                            error = error1;
                            pop_scene(scene_index);
                        } else {
                            pop_scene(scene_index);
                        }
                    });
                }
            }
            collections.forEach(changeColor)
        }
        callback(null)
    });
```

Figure 12 - The function used to update an imported collection

When a calendar is uploaded through the 'put.py' function its description is left empty, this could be used as an opportunity, if the uploaded calendar were intended to auto-update, its description could contain the URL it's updating from

alongside a short keyphrase to indicate that it's auto updating. This was done by first reading the input from the dropdown box created in the input scene into a variable, and then passing said variable into the colour changing function alongside the URL the calendar was imported from. Given the colour changing function was already designed to find the previously imported calendar's collection and adjust its values, it made sense to use this feature and adjust the description value as well. A variable was created to store the description of the calendar, setting it to nothing if the calendar wasn't auto-updating, but if it was, setting it to a string of 'Importing From:' followed by the URL passed into the function. After this, when the colour was adjusted in the function, the description was also updated, set to this new description variable. From here a python script could be made to update calendars with 'Importing From:' as the start of their descriptions. Because this function was now updating more than just the collection's colour, it was renamed from 'color_collection' to 'adjust_imported_collection'. This finally fully satisfied requirement four.

### 5.5 Implementation of Requirement Six

The next step needed to complete the project was finding each calendar that's description started with 'Importing From:'. The information included in and about each calendar are stored in folders in the same directory, thus finding the descriptions of specific calendars would be as simple as searching all the files within this directory for information about the calendars. Each of these folders is formatted in such a way that the collection for the given calendar (information that would include the collection's description) is stored as a dictionary within a .props file. To find all the calendars intended to automatically update then, the program would need to search through all these .props files, checking the contained descriptions, and then performing the update if this description is 'Importing From:'.

```
import time, os, json
from datetime import datetime


dirList = os.getcwd().split("\\")
collectionsDir = dirList[0] + "\\" + dirList[1] + "\\" + dirList[2] + "\\radicale\\collections"

while True:
    now = datetime.now()
    if (now.minute % 30) == 0 and now.second == 0:
        print(now.strftime("%H:%M:%S"))
        for r, d, f in os.walk(collectionsDir):
            for file in f:
                if file.endswith(".props"):
                    props = (open(os.path.join(r, file), "r")).read()
                    try:
                        propsDict = json.loads(props)
                        calDesc = propsDict["C:calendar-description"]
                        if (calDesc[:16] == "Importing From: "):
                            print(propsDict["D:displayname"])
                            print(calDesc[16:])
                    except:
                        False

    time.sleep(1)
```

Figure 13 - The contents of 'update.py'

To do this check, a new file was created which imported the os python library, this would allow it to search through all files within a given directory. The directory the project would search through, the one containing all the calendars, could be found by taking the base of the project's current working directory, and then appending '\radicale\collections' to it. After this, a for loop could be created to search each file in this directory, if the file were a .props file, its contents would be read, and turned into a dictionary using python's json library. With this dictionary the calendar's collection's description could be checked, for now, if it started with 'Importing From:' it would simply output the calendar's title and the URL it would be automatically updating from. This checking the description was contained in a try/except statement to ensure that calendars without a description wouldn't cause any errors for the program.

The final parts of requirement six that wasn't yet fulfilled was that this file check should begin at the same time as the main program was started, happening alongside it, and run on a consistent loop. To allow this file to run synchronously with the main program, the '__main__.py' file's code was adjusted, using the os library's 'spawnl' function to execute the code from the new 'update' file in parallel to the main Radicale program's functioning. As '__main__.py' is the first file called in the program, this means that as soon as the program starts, the auto-update check will start alongside it. Finally to make this code run on a consistent loop, the 'datetime' and 'time' libraries were imported. The directory searching for loop  was put inside an if statement that would run when the current minute number could be divided by 30 without a remainer, and when the current second number was 0, and then this if statement was put within in a while true loop that would check the current time, run the for loop if the if

statement's conditions were met, and then wait for a second before checking the time again. This would cause the file check to run twice an hour at half past the hour and on the hour exactly.

### 5.6 Failure of Implementation of Requirement Seven

As the project could now find all the calendars that required updating, doing so on a loop every thirty minutes, the final task would be for these calendars to actually be updated, thus fulfilling requirement seven. The first attempt at completing this goal was through importing the put function into the update file that had been created, this imported function could then be called for each calendar intended to update, passing in the same calendar URL and href as before so as to update not replace, and not break anything by changing the link. This could then be used in conjunction with the 'os.rmdir()' function to delete the contents of the calendar before it was reimported, allowing the contents to be replaced rather than duplicated (or more likely causing an error).

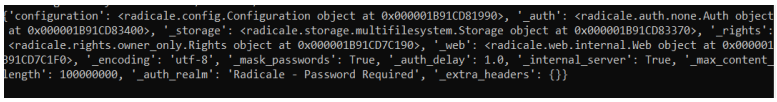



Figure 14 - The first level of contents for one of the five variables passed into the do_PUT function

This solution however, unfortunately didn't work. While the put function would be able to run and re-import the calendar as required for the task if given the correct inputs, the inputs it required were formatted in such a way that it would be very difficult to replicate calling it from a python file. Additionally, it would also be very difficult to replicate the actual information being passed to the function itself, much of it being dependent on the environment it was called from. Because of these issues, it became clear that for this project to auto update, it must be called from a javascript file.

As the update must be called from a javascript file, one was created, this file simply replicating the function previously created to upload the URL the user provides the project. This too hit an unfortunate snag. While it's possible to directly call a javascript file or function from a python file, this can only be done by using python to call javascript from the command line. At first this doesn't seem like too much of an issue, however javascript cannot normally be called from the command line, only able to do so if the user has some other program installed, such as the Node.js runtime; if this project were a unique project this would be fine, however as stated in requirement nine, this program must not use anything that cannot be assumed by the core program itself, this restricts it from relying on node to call this file. Furthermore, in tests to try and let this code function at all, using node as a stand-in before another workaround could be found, the project was still unable to correctly call the put function as intended, it seeming likely that the server element was also required.

Despite best efforts in attempting to find work-arounds, creating a html file to call the javascript and be called from python, attempting to adjust the original code written within 'fn.js', allowing it to be called using node, attempting to recreate the update loop within both a separate javascript file, and 'fn.js', the deadline fast approaching, and thus requirement seven remained incomplete.

## 6 Results & Discussion

### 6.1 Requirements Checklist

As mentioned in the implementation section, all but one of the requirements outlined in section 3.2 have been fulfilled. Requirement eight, the automatic updating of calendars was unable to be discussed for reasons mentioned in the implementation section. This is further expanded on in section 6.3 of this report.

| Requirement | Completed? |
|---|---|
| The user must be able to input a string into the program that the program can store. This basis must be established before the rest of the project is worked on to ensure that users are able to input URLs for the project to read. | Yes |
| The project must be able to upload the contents of a calendar stored at a URL, from a URL that it provided to it. Again, this is a requirement for the rest of the project to function at all, so must be completed before other steps are done. | Yes |
| After these first two steps are completed, the project should then be able to take a URL submitted by the user, and upload the calendar stored at this location, creating a collection from it. | Yes |
| The user must be able to input some details about their calendar that are then stored in the calendar's collection, specifically they should be able to select a colour to appear next to the collection, and should be able to select how exactly they want to import the calendar, either automatically updating (though not allowing them to directly edit it), or simply creating a copy of the calendar onto the Radicale server. | Yes |
| While importing a calendar the user shouldn't be able to interact with the rest of the web interface, being shown a loading screen. Additionally, after the calendar is imported they should be returned to the collections scene where their newly imported calendar is displayed. | Yes |
| The project must be able to search through all the collections on a consistent basis, checking if the calendar has been set to automatically update, this should occur synchronously with the program itself and not interrupt its processes. | Yes |
| If the project finds a calendar collection that has been set to automatically update, it should attempt to re import the calendar from the URL, maintaining the same href. If the import is successful, the calendar's old contents should be deleted. | No |
| This must all function without error, and if an error is caused for some reason (e.g. the user inputting an invalid URL) error details must be displayed without causing the program to stop functioning. | Yes |
| The project must not use any libraries or functions that aren't proven to already exist within the Radicale code, or aren't pre-installed with the language they're being used in. Additionally, the structure of any variables (primarily lists and dictionaries) shouldn't be adjusted. This is done to ensure that were the project to be merged into the Radicale repository it wouldn't break the program for any pre-existing users. | Yes |
| In a similar vein to requirement 9, the formatting and structure of the Radicale codebase should also be maintained within the project. If the project is intended to work with an existing program, it should be structured in a way that doesn't stand out and feels like a part of the original program rather than some extra code that's been tacked on to add functionality. This is discussed further in section 4.3 of this report. | Yes |

### 6.2 Performance Evaluation Discussion

Of the twelve parts of performance evaluation outlined in section 4.3, all but one have been completed, that one being evaluation 8, not completed due to failing to meet the requirement which that evaluation was intended to test.

| Test | Completed? | Proof |
|---|---|---|
| A calendar that already has an event will be imported from a given URL, providing no colour for the collection and importing it in a way that it doesn't automatically update. This will confirm that at the very least the calendar's title has been read from the URL. | Yes | LINK |
| This calendar will be imported into Mozilla's Thunderbird application to check that its events have been correctly imported into Radicale. | Yes | LINK |
| Another calendar will be imported into into Radicale that also isn't intended to auto update, this time the colour provided will be #FFFF00 (or 255, 255, 0) to confirm that the colour inputted is what the program is storing. The image manipulation program GIMP will then be used to confirm the collection's colour. | Yes | LINK |
| A calendar will be imported into Radicale that will attempt to automatically update. This calendar's collection's .props file will then be checked to confirm that the intent for this calendar to automatically update has been saved. | Yes | LINK |
| A non-URL string, 'TestString', will attempt to be imported through the web interface. This should cause an error message, but for the program to continue functioning. | Yes | LINK |
| A URL that doesn't have a calendar stored at it, 'https://www.google.com', will be imported through the web interface again expecting an error message but for the program to continue functioning. | Yes | LINK |
| The system that checks for auto-updating calendars will be temporarily adjusted to output the names of each calendar that are to be automatically updated, confirming that these calendars are being correctly identified for updating. | Yes | LINK |
| An automatically updating calendar will be imported into Mozilla Thunderbird, an adjustment will be made to the calendar wherever it's being imported from. The calendar will then be updated in Radicale, and Thunderbird will be refreshed to check for the adjustment. | No | N/A |
| During all the previous imports, attention will be kept to ensure that a user wouldn't be able to make any changes as a calendar is being updated. | Yes | As seen in the other videos, a loading screen is displayed whenever a calendar is being imported, preventing users from making changes. |
| | | |

| | | |
|---|---|---|
| A new calendar will be created on Radicale to confirm that this project hasn't broken that function of the program. | Yes | LINK |
| A calendar will be uploaded to Radicale from a file for the same reasons outlined in performance evaluation ten. | Yes | LINK |
| A raspberry pi with Radicale on it will be set up and use this project's code to import a calendar from a URL on it, confirming that this project, like the original program, can function on less powerful devices. | Yes | LINK |

### 6.3 Results Discussion & Evaluation

As can be seen in the requirements checklist and performance evaluation, nearly all of the original objectives and requirements for this project have been completely fulfilled. The project is a mostly successful addition to the original program, allowing the user to input a calendar from a URL, they can select specific information that will be stored in that calendar's collection, and the process by which they do so finishes requests before allowing users to do anything else, preventing users from breaking the system. It also functions overall without error, invalid user inputs are caught with an error message being displayed and the program continuing to function. Additionally the project maintains the structure and formatting of the Radicale codebase, variable and function names in the project following the lead set by the project, and all of the code functions entirely reliant on libraries that come with python or can be assumed from the rest of the code.

Unfortunately, the automatic updating of calendars that the user wanted to automatically update was unable to be programmed. While the user can select a calendar they wish to auto-update, and code will run very thirty minutes finding the calendars that should be updated, due to the fact that python cannot call javascript functions easily, an issue exacerbated by the constraints of the project, and the time limitations imposed, the actual updates will not occur. While this is a large failure for the project itself, the true impact of this from a user perspective isn't as much as it may at first seem.

As discussed in section 2.2, the use of this automatic updating would be that users can store all their calendars in one place, however due to how the updating works, it would still have required them to open the calendar being imported from if they needed to write to the calendar. The URLs for the calendars that would be imported into Radicale can instead, be directly imported into Thunderbird or similar applications, to a largely similar effect. The calendars still being stored only in these other applications is unfortunate as there is still reliance on the continued functioning of proprietary calendars for these updating calendars to continue to function, while this project was intended to somewhat aid with this issue, the failure to do so isn't a catastrophic failure.

Another part of the project that's worth discussing is the storing of URLs in each collection's description. While this workaround functions fine, especially due to the constraints of not being able to change the formatting of the collection files, there are still a few issues with it. Firstly the fact that it means the user is unable to have a description for calendars that are set to automatically update, secondly, the updating system could be broken if a calendar not intended to automatically update were to have a description starting with 'Importing From:', or rather it could if the updating system were fully implemented. The final issue with storing the files in this way is the potential security risk it poses, as Google correctly brings attention to in the clips linked in section 6.2, the private address of a given calendar shouldn't be shared. Storing the URL in such an easily accessible way as in the descriptions of

these calendars therefore makes this private address much more easily accessible, a definite risk given what private information can be stored within these calendars.

# 7 Conclusion

In conclusion, while one project requirement was unable to be fulfilled due to limitations in language and time, the overall project was a success. The addition of URL based importing to the existing Radicale codebase is a great achievement, the specific implementation of the ability is also to be applauded, working fully within the existing constraints of the Radicale program, both the technical constraints of libraries, and the structural constraints of comments and spellings.

If this project were to continue being developed, there's two core areas that have scope for future work. Firstly the completion of the project's automatic updating of calendars, this would also include adjusting where the URL is stored for these calendars so as to pose less of a security risk. And secondly once that were completed, the ability to back-edit calendars that are intended to auto-update, allowing users to somehow edit their proprietary calendars from their open source calendar applications, making adjustments to the imported Radicale calendars.

# 8 References


[1] *Radicale V3 Documentation* (no date) *Radicale v3 Documentation*. Kozea. Available at: https://radicale.org/v3.html (Accessed: April 18, 2023).

[2] Choi, J.P., Jeon, D.S. and Kim, B.C., 2019. Privacy and personal data collection with information externalities. *Journal of Public Economics*, *173*, pp.113-124.

[3] Streitfeld, D., 2012. Google is faulted for impeding US inquiry on data collection. *The New York Times*.

[4] Farooqi, Y.S. and Baig, W., 2017. Life span of social media In last ten years. *Journal of Mass Communication Department, Dept of Mass Communication, University of Karachi*, *16*.

[5] Celosia, G. and Cunche, M., 2020. Discontinued privacy: Personal data leaks in apple bluetooth-low-energy continuity protocols. *Proceedings on Privacy Enhancing Technologies*, *2020*(1), pp.26-46.

[6] Smith, S., 2001. Microsoft and the European Union Face Off Over Internet Privacy Concerns. *Duke L. & Tech. Rev.*, *1*, p.1.

[7] Nas, S., 2019. Data Protection Impact Assessment: Assessing the Risks of Using Microsoft Office ProPlus. *Eur. Data Prot. L. Rev.*, *5*, p.107

[8] Vaughan-Nichols, S. (2011) *Freedom box: Freeing the internet one server at a time*, *ZDNET*. Available at: https://www.zdnet.com/article/freedom-box-freeing-the-internet-one-server-at-a-time/ (Accessed: February 1, 2023).

[9] *.Lock file extension* (2019) *LOCK File Extension - What is a .lock file and how do I open it?* FileInfo.com. Available at: https://fileinfo.com/extension/lock (Accessed: May 7, 2023).


# 9 Appendices

## 9.1 Reflection

During this project there are a few areas in which I believe I could have improved, firstly, my time management skills weren't the greatest throughout. While I was able to complete the core functionality of the project, I unfortunately wasn't able to leave enough time to complete the automatic updating functionality, consistently underestimating the time that it would take and ultimately leaving myself until very near the deadline set in the objectives to finish it. This is a mistake that I'll ensure I don't repeat, leaving more time than needed for future projects and not delaying work on these parts of the project as I consistently did throughout this due to assuming things would take less time than needed and putting them off.

Another failure of this project was that I made some of the requirements slightly too broad or containing multiple parts when it would have been easier to split them. Requirement four for example is the requirement that the user is able to input information, colour and the import type, into the program. This could already be sectioned into separate requirements, one being the requirement that the user can input colour, the other than the user can input the import type, making it clearer during implementation what requirements had been fulfilled by a section, rather than spending multiple sections implementing a 'single' requirement. Another problem with this specific requirement however comes in its lack of clarity, while it asks that the user can input those to be stored in the collection, this presupposes an interface for the user to input the colour and import type, essentially making this seemingly single requirement in actuality three separate sections.

Despite these failures there are a few things I learnt from this project that I'm glad of. Before starting on this project I had only very basic knowledge of HTML, and no knowledge of javascript at all. Through this I've been able to learn more about both languages, and learn more of how different languages interact and can work together, an aspect of computer science that I'd not explored before, having primarily worked on single language programs, or programs that used very little of other languages.

Additionally, this project gave me the chance to work with an existing codebase rather than building a program from scratch. While the latter is useful, knowing I have the ability to do the former has made me a lot more confident in my own capabilities, and is a skill I know will be extremely useful to have when working in industry.

### 9.1 Link to Code

Link to the project codebase