

Worldwide Customer Services

Embedding WebFOCUS in a Python Web Application

WebFOCUS 8206

August 2019



Active Technologies, EDA, EDA/SQL, FIDEL, FOCUS, Information Builders, the Information Builders logo, iWay, iWay Software, Parlay, PC/FOCUS, RStat, Table Talk, Web390, WebFOCUS, WebFOCUS Active Technologies, and WebFOCUS Magnify are registered trademarks, and DataMigrator and Hyperstage are trademarks of Information Builders, Inc.

Adobe, the Adobe logo, Acrobat, Adobe Reader, Flash, Adobe Flash Builder, Flex, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Due to the nature of this material, this document refers to numerous hardware and software products by their trademarks. In most, if not all cases, these designations are claimed as trademarks or registered trademarks by their respective companies. It is not this publisher's intent to use any of these names generically. The reader is therefore cautioned to investigate all claimed trademark rights before using any of these names other than to refer to the product described.

Copyright © 2019, by Information Builders, Inc. and iWay Software. All rights reserved. Patent Pending. This manual, or parts thereof, may not be reproduced in any form without the written permission of Information Builders, Inc

CONTENTS

Table of Contents

CONTENTS	3
EXECUTIVE SUMMARY	5
Roles	5
Application Overview	6
Setting Up the Flask Application	8
Setting Up WebFOCUS Environment for RESTful Web Services	10
Cross-Origin Resource Sharing (CORS)	10
Reporting Server	11
Authentication: Sign In and Sign Out	13
Sign In Steps	13
wfrs.py – Session Object	14
Session Object Initialization	14
Session Object Sign On	15
Session Object Sign Off	16
app.py - WebFOCUS Authentication in our Flask Application	17
Sign On	17
Sign Off	18
Making Requests	19
List Files in WebFOCUS Path	21
Listing Files from a Path Steps	21
app.py – List Files in Path	21
Get Files in Path as XML Response	21
Create Python List of File Names from XML Response	24
HTML & Jinja2 – Creating a Drop-down List of Report Names	24
Run WebFOCUS Report	27
Procedure: How to Run a WebFOCUS Report	27
Running and Retrieving a WebFOCUS Report	27
Creating an Interface to Display WebFOCUS Reports in Flask App	30

Run_reports.html - Creating an iFrame to Display WebFOCUS Report	31
ReportCaster Schedules	32
Running a ReportCaster Schedule	32
Creating a Simple Table of Schedules.....	33
Get Schedule Info and Log	40
Procedure: How to Schedule Information and Log Retrieval	41
Submitting Schedule Name	41
Retrieving Schedule Information.....	43
Retrieving Schedule Job Log	47
Run Report Deferred	52
Procedure: How to Run a Deferred WebFOCUS Report.....	52
Run a Deferred WebFOCUS Report in the Flask Application.....	53
Defer_reports.html - Creating a Form to Request a Deferred WebFOCUS Report and Outputting Result Messages.....	54
Submitting a Request for a Deferred WebFOCUS Report	55
Processing Responses for a Deferred WebFOCUS Report	56
View and Use Deferred Tickets	57
App.py - Submitting a Request for a Deferred WebFOCUS Report.....	58
Deferred_reports_table.html - Creating a Table of Deferred Reports.....	60
Deferred_reports_table.html - Creating a View Item Form.....	62
App.py - Getting a Deferred Report	63
Delete Item.....	64
App.py - Delete Item.....	64
Deferred_reports_table.html - Delete Item in Deferred Reports	65
WebFOCUS Report Static CSS/JS Files.....	66

EXECUTIVE SUMMARY

This document walks through a sample web application that allows you to utilize WebFOCUS features without directly using the WebFOCUS client. The web application, written in Python using the Flask framework, interacts with WebFOCUS using RESTful web service calls and displays the information to the user.

The Python application allows you to interact with three major features of WebFOCUS:

- Running reports directly.
- Running a report deferred.
- Running a report schedule using ReportCaster.

Roles

The following list shows the contributors and support resources who helped deliver this project.

Worldwide Customer Services

Name	Title
Hamza Qureshi	Intern
Emily Nesson	Intern

Other Information Builders Resources

Name	Department	Title
Alan Boord		
Jessica Chen		
Frances Gambino		
David Glick		
Ira Kaplan		
Stefan Kostial		
Tony Li		

Breda McCrohan		
Alain Theriault		

Application Overview

Important Note: This document assumes the reader knows the basics of WebFOCUS RESTful Web Services. If this is not the case, please refer to https://webfocusinfocenter.informationbuilders.com/wfdesigner/pdfs5/embedded_bi_8205.pdf for further information.

This application uses the Python Flask web framework to create a web server. The server interacts with end users through its front end, and interacts with WebFOCUS through REST calls in its back end.

The Flask app will use the third-party Python Requests library for its Session object, which will be used to make HTTP requests to WebFOCUS RESTful web services.

This document assumes some prior knowledge of REST, Python, requests, and Flask.

In particular, it is important to know the basics of how a Flask application processes a request it receives, and how to utilize the requests library to create a request and process its response.

Most importantly, application and request contexts for flask are used in the design decisions for this application, so you may find more information in Flask's documentation, found here: <https://flask.palletsprojects.com/en/1.1.x/>

A single file is used for the Flask application, app.py, in the directory Embedded_WF. The Embedded_WF directory will contain a folder called Templates where the html code will be.

A wfrs.py file is also created to store the independent WF_Session class that will be created to initialize WebFOCUS requests. This will be imported into app.py.

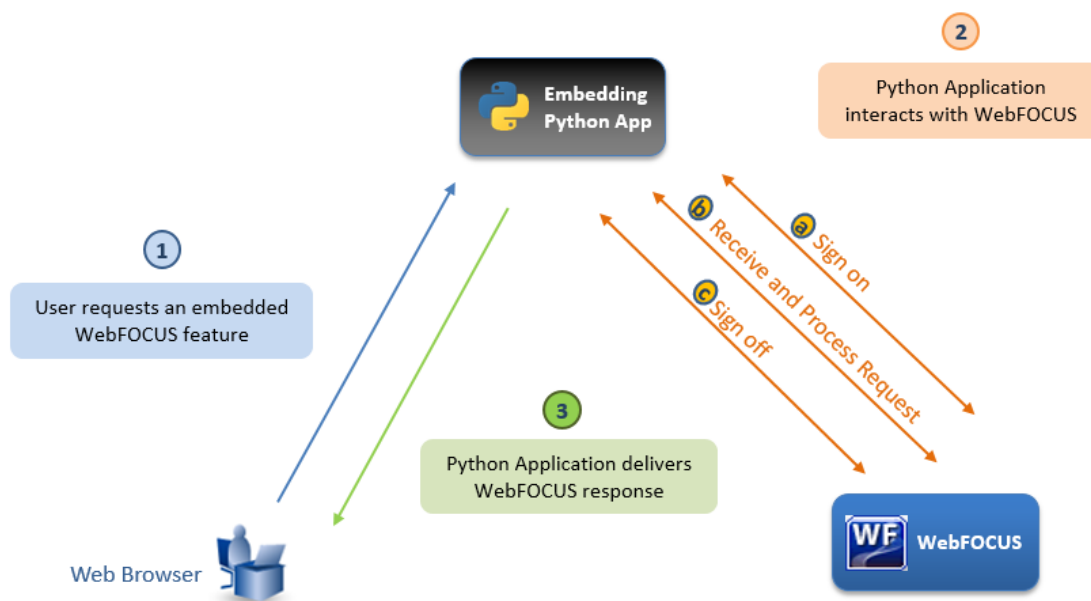
The templates will be rendered in Flask through the jinja2 templating language. This allows rendering of variables into a static HTML to be sent to a client. For example, if you want to create a static HTML table of a schedule's log, you can use jinja2 in the .html file and pass in Python variables through Flask, where they will be properly rendered. This avoids issues of rendering HTML as strings through Python.

The code for this documentation is hosted at
<https://github.com/Hamza-Q/IBI-WebFOCUS-REST-Sample>

Information Builders takes no responsibility for the correctness of this code base beyond the publish date.

The overall architecture for this application is as follows:

Flow for Python Web Application WebFOCUS Connection



To the end user operating the web browser, their own connection to the WebFOCUS client does not impact their experience. Everything related to WebFOCUS is done via the Python application so only the Python procedures in (1) and (3) directly interact with the web browser.

This architecture holds if the Python application is on a public network while the WebFOCUS client is behind a private network or firewall, which is why it was selected for this sample application.

Setting Up the Flask Application

Flask is a web server microframework, so it is easy to get set up. It is strongly recommended to follow the official Flask tutorial first if you do not have experience with Flask.

Flask and third party application requests will be used while running Python 3.7.3. Note that Python 3.6+ is supported. To install flask and requests, it is highly recommended to use pip with the commands:

```
pip install flask
```

```
pip install requests
```

All Flask codes will be stored in the app.py file.

The following image shows a sample Flask syntax.

```
from flask import Flask, render_template, request, session, \
    url_for, redirect, flash, g, send_from_directory, \
    make_response, send_file, abort

# Initialize app
app = Flask(__name__)

# TODO: Extract these from config file or env vars
ibi_client_protocol = "http"
ibi_client_host = "localhost"
ibi_client_port = "8080" # standard is 80 for http and 443 for https
ibi_rest_url = \
    f'{ibi_client_protocol}://{ibi_client_host}:{ibi_client_port}/ibi_apps/rs'
ibi_default_folder_path = "WFC/Repository/Public"

if __name__ == '__main__':
    # secret key can be randomly generated via commandline:
    # python -c 'import os; print(os.urandom(16))'
    # TODO: Add security; should import from config file or env variable
    app.secret_key = '<insert secret key here>'
    app.run(host='localhost', port=5000, debug=True)
```


A production Flask application should include better security and precautions, but for this example, the focus is on interacting with WebFOCUS in the back end. Most of these variables should be set up as environment variables or inside a config file.

Many libraries will be imported later in this application, but for now, all that is needed is Flask.

lbi_client variables will relate to the WebFOCUS client it is interacting with. It does *not* need to be on the same host or port as this Flask application, but the server the Flask app runs on must be able to access the WebFOCUS client. Since there is a flask application, you can use its features to your advantage when embedding WebFOCUS.

In Flask, the concept of contexts is fundamental to designing the application. Variables do not persist between requests, sessions, and application creation unless specifically requested to, and even then there are limitations.

In particular, Flask has a 'g' object, which is created when the application is initialized to receive a request, and a 'session' object, which is a dictionary, based on the session cookie containing JSON serializable variables (distinct from the requests. session or the WF_Session object. For more information, see the official Flask documentation.

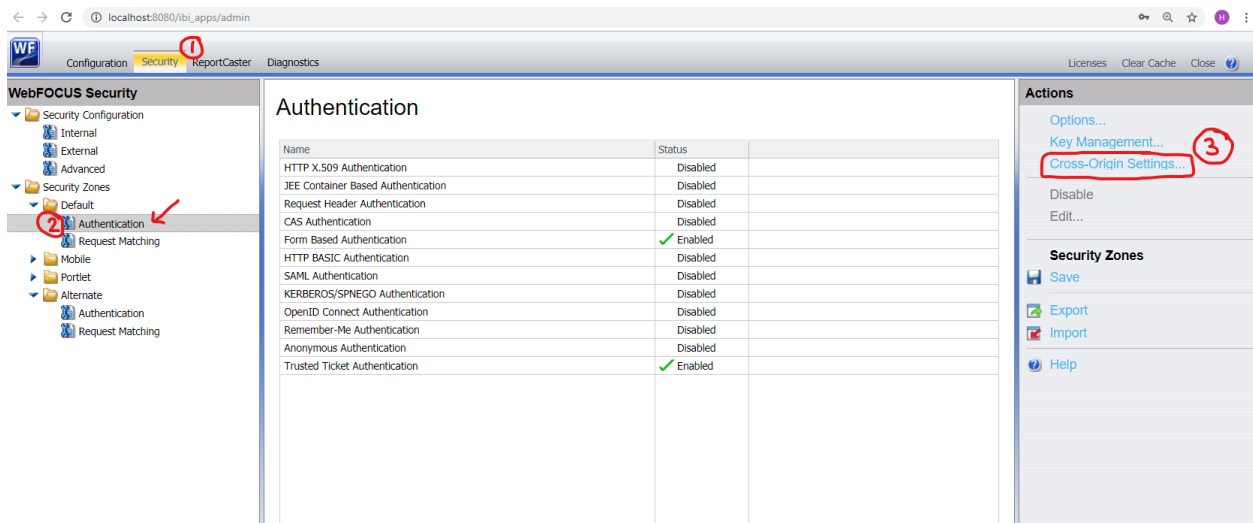
These will often be used, as you will see in the Authentication section.

Note: By using the host 'localhost', Flask by default only allows your local machine to access the server. If you want the application accessible to other device on the network, setting the host to '0.0.0.0' will use your hostname to be visible. However, this should ONLY be used for development purposes; for all production uses, a production server such as Apache Tomcat should host the server.

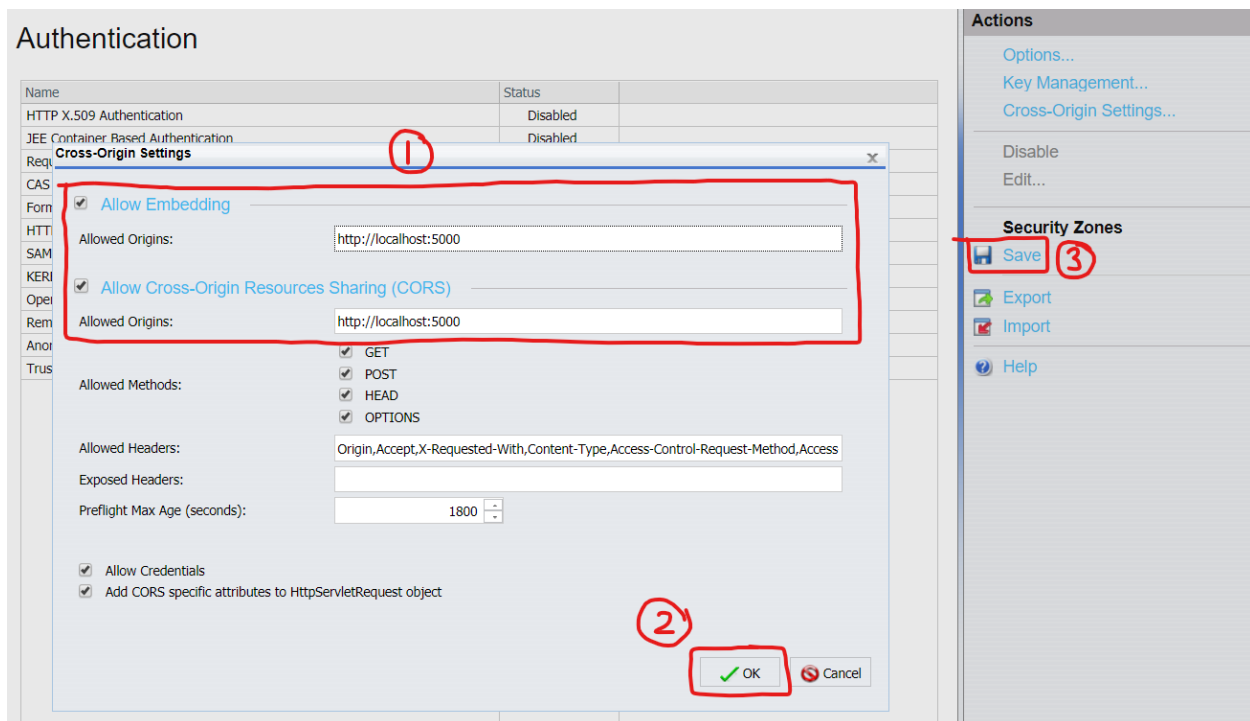
Setting Up WebFOCUS Environment for RESTful Web Services

Cross-Origin Resource Sharing (CORS)

To access WebFOCUS Resources from a different protocol, host, or port number, you must enable Cross-Origin Resource Sharing (CORS) on WebFOCUS. To do so, follow the numbered steps in the following images of the administration console of WebFOCUS:



After you click on Cross-Origin Settings, the following menu appears. Select *Allow Embedding* and *Allow Cross-Origin Resources Sharing (CORS)* then fill in the blanks with the URL of your web application. In this case, it will be <http://localhost:5000>. Then, click on *OK*, and *Save* under Security Zones.



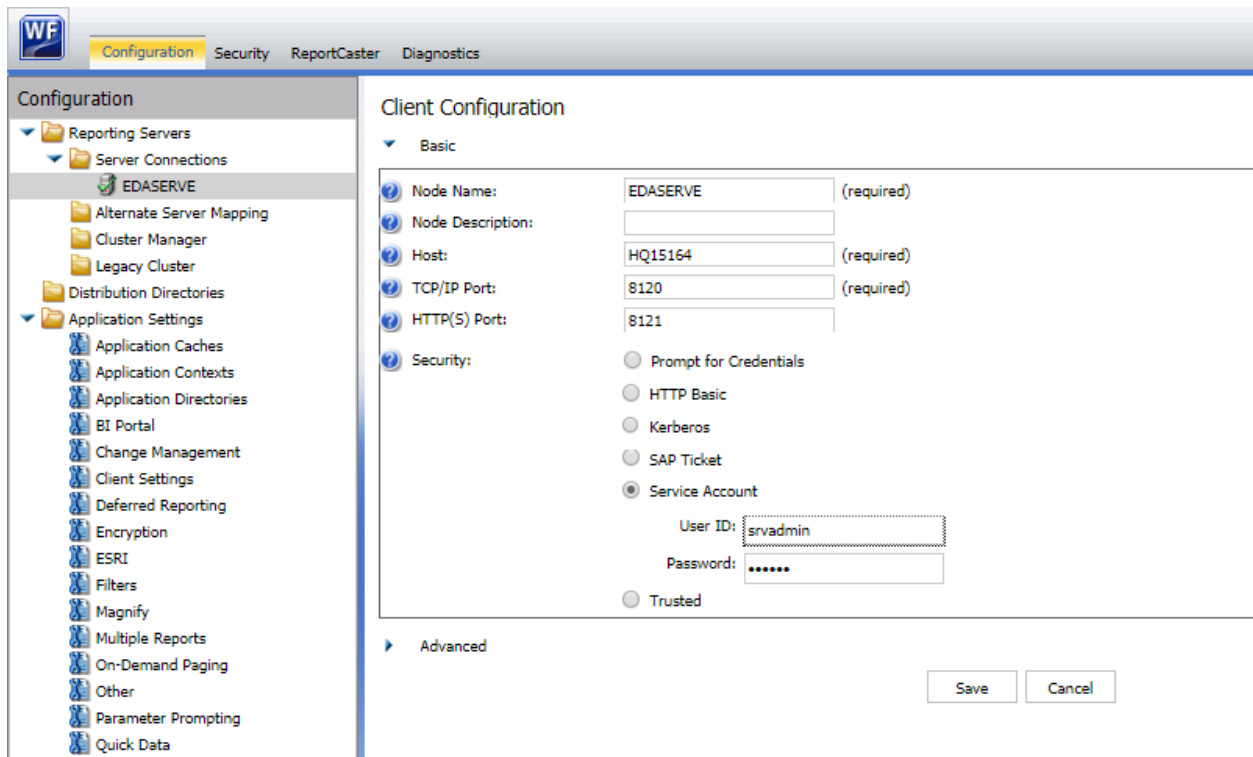
Finally, to see changes, restart your Tomcat server hosting WebFOCUS under Windows Services.

Reporting Server

WebFOCUS requests made to the client still require proper authentication to access Reporting Server content.

In REST requests to the WebFOCUS client, the parameters “IBIRS_username” and “IBIRS_password” can be used to authenticate for the Reporting Server.

For the purpose of this sample, we will use service account credentials to authenticate automatically with our REST requests. This can be set up in the WebFOCUS Client Administration Console. Under the Configuration tab, Reporting Servers/Server Connections/[Node name], select “Service Account” under the “Security” field and enter Reporting Server credentials as shown below:



A service account for this client can be configured on the Reporting Server.

If you do not consider Reporting Server security, requests will be redirected to the Reporting Server sign in page, an undesirable response.

Authentication: Sign In and Sign Out

Before you can embed WebFOCUS in the application, you must understand how to make authenticated requests to WebFOCUS.

This section will show the Python code generating requests made to sign in on the current application to WebFOCUS.

Sign In Steps

Sign in involves:

1. Making the request to sign in through WebFOCUS RESTful web services.
2. Confirming sign in was successful.
3. Saving the WF-JSESSIONID cookie from the HTTP response and saving the CSRF Token generated in the XML response body.

To do these steps, the Python Requests module will be used, specifically the session object from this to store the WF-JSESSIONID cookie and CSRF Token for future requests.

wfrs.py – Session Object

Session Object Initialization

In the wfrs.py file, you will initialize a Session object which inherits from requests.Session. It will have additional functionality to sign on and sign off from WebFOCUS, and save the CSRF Token to be used in other WebFOCUS requests.

```
import requests

class WF_Session(requests.Session):
    def __init__(self):
        requests.Session.__init__(self)
        self.IBIWF_SES_AUTH_TOKEN = None
```

You must initialize the parent requests.Session object in the __init__ method.

The self.IBIWF_SES_AUTH_TOKEN = None line is a placeholder for the instance's CSRF Token, which will be retrieved after sign in.

You can now use this object to sign in.

Session Object Sign On

You can create a method in the WF_Session object, mr_sign_on, which signs in to WebFOCUS using a REST call.

It takes the protocol, host, port, userid, and password as parameters, then sends the request using these values.

The response will contain an XML of whether the connection was established, or if there was an error in the login.

This method parses the XML response to find the CSRF token, then saves it to the WF_Session object's IBIWF_SES_AUTH_TOKEN attribute.

```
def mr_sign_on(self,
                protocol='http',
                host='localhost',
                port='8080',
                userid='admin',
                password='admin'):
    """WebFOCUS Repository: Authenticating WebFOCUS Sign-On Requests."""

    # Stored for sign off
    self.protocol = protocol
    self.host = host
    self.port = port

    data = {
        'IBIRS_action': 'signOn',
        'IBIRS_userName': userid,
        'IBIRS_password': password,
    }
    url = '{}://{}:{}/ibi_apps/rs/ibfs'.format(self.protocol,
                                              self.host,
                                              self.port)

    response = self.post(url=url, data=data)
    self._save_ibi_csrf_token(response.content)
    return response
```

You can use the `xml.etree.ElementTree` library to parse the XML response. To do this, you must add another import statement at the top of the `wfrs.py` file:

```
import xml.etree.ElementTree as ET
import requests

class WF_Session(requests.Session):
    ...
```

The code for `_save_ibi_csrf_token` is as follows:

```
def _save_ibi_csrf_token(self, xml):
    """Save IBI_CSRF-Token_Value from response to sign-on request."""

    tree = ET.fromstring(xml)
    token = tree.find('properties/entry[@key="IBI_CSRF-Token_Value"]')

    token_value = token.attrib['value']
    self.IBIWF_SES_AUTH_TOKEN = token_value
```

It utilizes friendly user input, where you can add try/except blocks, and return a bool True/False to let `mr_sign_on` know if it was successful.

When a `WF_Session` instance is created and the `mr_sign_on` method is called with valid parameters, the instance should automatically set the proper `WF-JSESSIONID` cookie to successfully make future requests to WebFOCUS.

Session Object Sign Off

When a sign off call is made to WebFOCUS, the current `WF-JSESSIONID` will be invalidated, so no requests can be made without signing in again.

For security purposes, after requests are made and finished, you will eventually be signed out from WebFOCUS. The `WF_Session` object's `WF-JSESSIONID` cookie will be automatically invalidated and then set to a new, random value by the WebFOCUS client server. However, your CSRF Token was stored in the instance, so it must be set to *None*.

The following image shows the code for `mr_signoff`.


```
def mr_signoff(self):
    """WebFOCUS Repository: Signing-Off From WebFOCUS."""

    data = {'IBIRS_action': 'signOff'}
    url = '{}://{}:{}/ibi_apps/rs/ibfs'.format(self.protocol,
                                              self.host,
                                              self.port)

    self.IBIWF_SES_AUTH_TOKEN = None
    self.post(url=url, data=data)
```

app.py - WebFOCUS Authentication in our Flask Application

Sign On

You now have Python code to interact and authenticate with WebFOCUS in the form of a WF_Session object and mr_sign_on and mr_signoff method definitions.

However, these exist in the wfrs.py file. If it is located in the same directory as app.py, where your Flask app exists, you can import it through the statement:

```
import wfrs
```

In app.py, you can use the 'g' object to store a WF_Session instance for a request. This way, if you connect to WebFOCUS to process a user request, you can control how the WF_Session will be used between functions in the application. You only need to instantiate one WF_Session and sign on once to process every request to WebFOCUS as needed, before signing out and destroying the connection.

To do this, you can create a wf_login function to retrieve the WF_Session, creating an instance if it does not already exist or retrieve an already existing instance.

You can call the wf_login function in every Flask route and view where you need to connect to WebFOCUS, such as when retrieving a report or listing files in a path.

The wf_login function is as follows:

```
# Creates and returns a signed in webfocus session object
def wf_login():
    # g is the application context; g objects are created and destroyed
    # with the same lifetime as the current request to the server
    # By creating the WF Session within g, we can use one connection
    # per request and sign out at the end,
    # rather than sign in/out for every action

    # Create a WF Session if one does not already exist
    if 'wf_sess' not in g:
        g.wf_sess = wfrs.WF_Session()
        g.wf_sess.mr_sign_on(
            protocol=ibi_client_protocol,
            host=ibi_client_host,
            port=ibi_client_port
        )
    return g.wf_sess
```

You can define the key for the WF_Session instance in 'g' to be 'wf_sess', so you can retrieve the instance through dot notation (g.wf_sess) or through other dictionary methods such as g['wf_sess'], g.get['wf_sess'].

The default credentials will be:

- username: **admin**
- password: **admin**

It is recommended to create a service account or utilize a different method of Single Sign-On such as a trusted ticket if those options are available to you.

Sign Off

Without any other code, the Flask 'g' object and the WF_Session instance with it will be destroyed after the request is processed and a view is returned. You can use a Flask decorator to sign off before it is destroyed for security purposes, as shown in the following image.

```
# signs out of WF after request (closes connection)
@app.teardown_appcontext
def teardown_wf_sess(error=None):
    wf_sess = g.pop('wf_sess', None)
    if wf_sess is not None:
        wf_sess.mr_signoff()
```

The `@app.teardown_appcontext` decorator before the defined function tells the Flask app that this function should be called right before it destroys the app context ('g').

You can retrieve your instance of the `wf_sess` and remove it from the 'g' dictionary, if it exists. If it does not, there is no need to sign out, otherwise, you will call the `mr_signoff` method.

The error parameter is passed in by Flask when this function is called, so that you can adapt your function to any errors processing the request. However, you must sign out of WebFOCUS when the request is finished regardless if there was an error or not, so you do not use it in the function.

Making Requests

To interact with WebFOCUS in the Flask application, you can now just use a simple flow of:

- Calling the `wf_login()` function and assigning the session to a variable (we use `wf_sess`).
- Preparing request URL, parameters, payload using proper REST calls.
- Making the request.
- Parsing the response
- Passing relevant data to the front-end html template

An important step to note is that when a POST request to WebFOCUS is made, such as to delete an item, the CSRF token must be included in the payload.

The `WF_Session` object inherits from the `Requests Session` object, so when given a dictionary of key value pairs as a `data` parameter, the body will be properly formatted accordingly. This makes including a CSRF token easy.

The following image is an example code snippet to delete an item within a Flask app route.

```
wf_sess = wf_login()
item_name = 'Report1.fex'
payload = dict()
payload['IBIWF_SES_AUTH_TOKEN'] = wf_sess.IBIWF_SES_AUTH_TOKEN
payload['IBIRS_action'] = 'delete'
response = wf_sess.post(
    f'{ibi_rest_url}/ibfs/WFC/Repository/Public/{item_name}',
    data=payload
)
```

If there exists an item named “Report1.fex” in the public repository, it will be deleted via this REST call.

Note: From now on, we will be working within app.py always.

List Files in WebFOCUS Path

Before you can access any files in WebFOCUS, you need to retrieve a way to identify and call them. WebFOCUS Repository items follow unique file paths which you can use.

In this sample application, all report and schedule files will be under the public repository, WFC/Repository/Public, which is why you initialized the `ibi_default_folder_path` to this when you created the Flask app.

If a stronger use case is required, the steps in this section can be extended to recurse on folders within repositories, but this will be not covered.

Listing Files from a Path Steps

- Making the request to list files in the specified path.
- Parsing XML response.

app.py – List Files in Path

Get Files in Path as XML Response

When the proper call is made, WebFOCUS will return an XML response object. If it's a success, the parent node will be an `IBRSResponseObject`. The `rootObject` is one of its children, which is what you are interested in.

The `rootObject` will contain all files in the path as children with the respective object type as a tag. For example, schedules will be tagged as “`casterObject`”, and reports will be tagged as “`FexFile`”.

The Python code to retrieve the `rootObject` XML as an ET root node is:

```

# gets xml ET object of response
def list_files_in_path_xml(path=ibi_default_folder_path):
    wf_sess = wf_login()

    params = {'IBIRS_action': 'list'}
    response = wf_sess.get(
        f'{ibi_rest_url}/ibfs/{path}',
        params=params,
    )

    files_xml_response = ET.fromstring(response.content)

    files_xml = files_xml_response.find('rootObject')

```

From the parent response object, you can retrieve the rootObject as files_xml.

You can further parse this rootObject and only return XML children of files of the type you wish.

The following code extends the function to list all the files of a certain file_type in this directory:

```
# gets xml ET object of response
def list_files_in_path_xml(path=ibi_default_folder_path, file_type=""):
    wf_sess = wf_login()

    params = {'IBIRS_action': 'list'}
    response = wf_sess.get(
        f'{ibi_rest_url}/ibfs/{path}',
        params=params, # data=payload
    )

    files_xml_response = ET.fromstring(response.content)
    files_xml = files_xml_response.find('rootObject')

    if file_type:
        # Check children and find those without proper return type
        invalid_children = []
        # Note: removing from files_xml within first for loops leads to errors;
        # for loop does not iterate over every child
        for child in files_xml:
            if child.get("type") != file_type:
                invalid_children.append(child)
        for child in invalid_children:
            files_xml.remove(child)
    return files_xml
```

If the function is called with no file_type selected, it will default to including all files.

Create Python List of File Names from XML Response

The XML Tree object is hard to use in Python code. To simplify, you can parse and extract its information into a more easily usable data structure. A list of the item names in the selected path (with the selected file type if needed) is generated for easier use in the Flask app.

The following function will take in the rootObject created above (parent directory) and return a list of the item names of its children (files in the directory):

```
def files_xml_to_list(files_xml):  
    item_list = []  
    for item in files_xml:  
        item_name = item.attrib.get("name")  
        item_list.append(item_name)  
    return item_list
```

HTML & Jinja2 – Creating a Drop-down List of Report Names

Using Flask's `render_template` function, you can pass in the name of an HTML template that is in the Templates directory. The Templates directory itself should be in the same directory as the WebFOCUS app.

In the following section, an interface is provided for users to view WebFOCUS reports, using the `/run_reports` route. So, if the app is hosted on 'http://localhost:5000', users can navigate to 'http://localhost:5000/run_reports' using any means and access this page.

You can call the HTML template for this page 'run_reports.html' and it will contain jinja2 syntax to dynamically generate a static html page.

On this page, a drop-down list of report names for the user to select is created.

The following image shows the HTML + jinja2 code for this.


```

<h2> Select a Report Name: </h2>
<form action="/run_report" id="run_report" target="report_frame" method="post">
    <select name = "report_name" id="report_name">
        {% for rep_name in reports %}
            <option value="{{rep_name}}">{{rep_name}}</option>
        {% endfor %}
        <input type="submit" value = "Submit" id="button" />
    </select>
</form>

<iframe src="about:blank" id="report_frame" name="report_frame"
    style="height: 100%; width: 90%;" />
</iframe>

```

The form will submit the option selected as a post request to `http://localhost:5000/run_report`. The html tag *name* determines the key of the request, and *value* determines its value, which will be used to receive and process in Flask.

The Flask route `/run_report` must process the request and return a proper HTML response to be displayed in the iframe.

However, first the template under the `/run_reports` route must be rendered, so that this interface is accessible. This can be done with the following code in `app.py`:

```

@app.route('/run_reports')
def run_reports():
    files_xml = list_files_in_path_xml(file_type="FexFile")
    # reports is a list of report names
    reports = files_xml_to_list(files_xml)
    return render_template('run_reports.html', reports=reports)

```

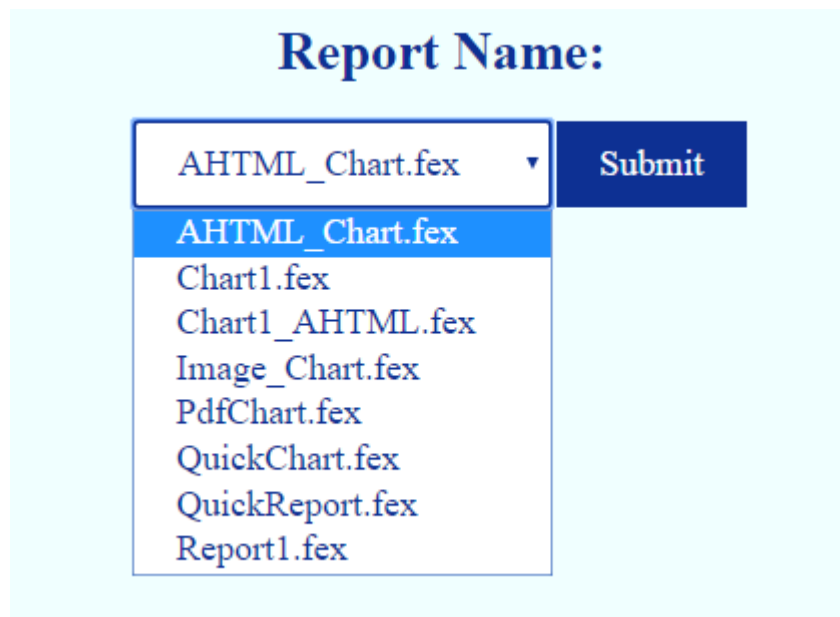
Using the functions coded above, send a list of report names to the jinja2 template in `run_reports.html`. In the HTML code, the following syntax appears:

```
{% for rep_name in reports %}
    <option value="{{rep_name}}">{{rep_name}}</option>
{% endfor %}
```

Jinja2 loops through the Python variable `reports` that was passed in, which is a list of report names.

It will create an option for the select drop-down list with names and values for each report name.

The HTML page when rendered on 'http://localhost:5000/run_reports' will look like the following image:



The image shows a web form with a light blue background. At the top, the text "Report Name:" is displayed in a large, bold, dark blue font. Below this text is a form element consisting of a dropdown menu and a button. The dropdown menu is currently open, showing a list of report names: "AHTML_Chart.fex", "Chart1.fex", "Chart1_AHTML.fex", "Image_Chart.fex", "PdfChart.fex", "QuickChart.fex", "QuickReport.fex", and "Report1.fex". The first option, "AHTML_Chart.fex", is highlighted in blue. To the right of the dropdown menu is a dark blue button with the word "Submit" in white text.

The following section goes through the process of receiving the report name when it is submitted and running the associated WebFOCUS report.

Run WebFOCUS Report

In this sample application, you can retrieve a report from the WebFOCUS public repository and display it using iFrame in the Flask application.

Requests are made to WebFOCUS to get the report, and the content of the report is read to return the proper response.

Procedure: How to Run a WebFOCUS Report

1. Make a login connection to WebFOCUS and a request to retrieve the report.
2. Determine the content type of the report.
3. Return the proper response based on the content type.

Running and Retrieving a WebFOCUS Report

When a call to run a report is made to WebFOCUS, a get request is created to retrieve the report name. If the report_name is not found in WebFOCUS, you are redirected to the run_reports page of the Flask application.

By authenticating a login to WebFOCUS, a connection is created between the Flask application and a call to WebFOCUS. After doing so, you can create a post request to WebFOCUS for the report that has been entered.

The WebFOCUS response contains a content type header that you check. Based on this content type header, you can properly format the Flask response, if necessary. For instance, if the report is an HTML file or an image, it is returned as so, respectively.

The following image shows the syntax for run report.

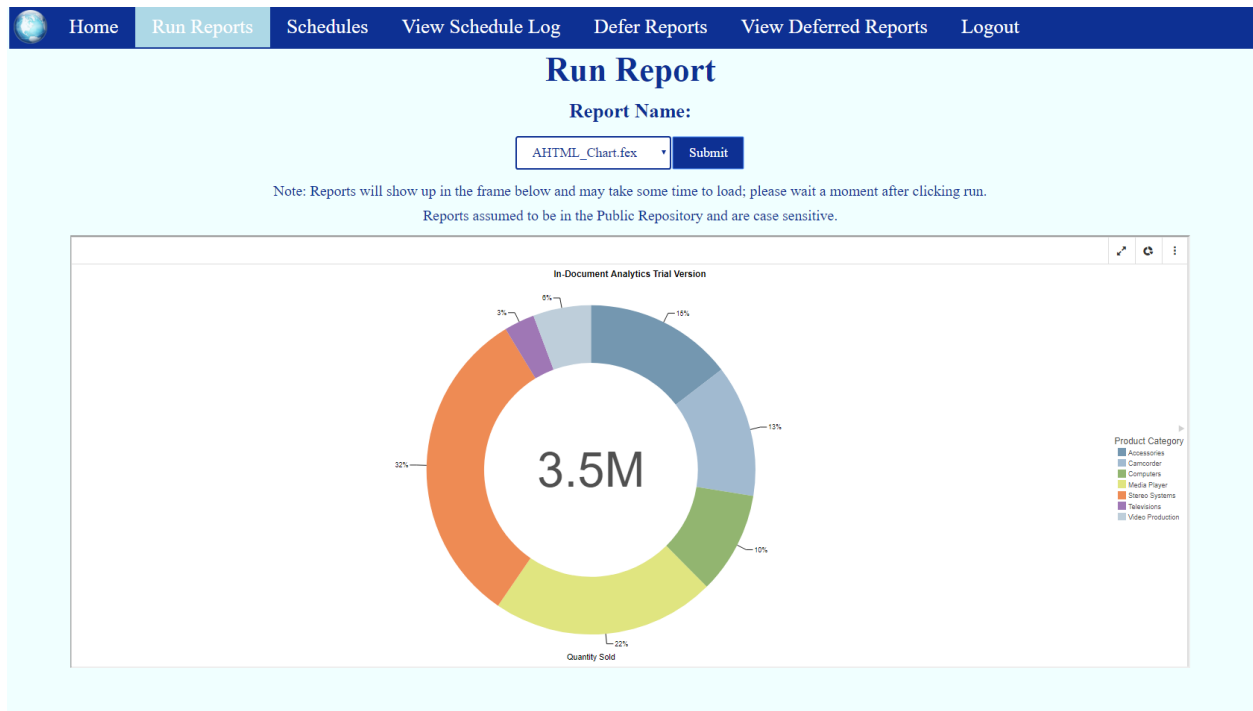
```

@app.route('/run_report', methods=['GET', 'POST'])
def run_report():
    report_name = request.form.get('report_name')
    if not report_name:
        return redirect(url_for('run_reports'))
    wf_sess = wf_login()
    # Used to properly return PDF and EXCEL files from WebFOCUS
    turn_off_redirection_xml = \
        '''<rootObject _jt="HashMap">
            <entry>
                <key _jt="string" value="IBFS_contextVars"/>
                <value _jt="HashMap">
                    <entry>
                        <key _jt="string" value="IBIWF_redirect"/>
                        <value _jt="string" value="NEVER"/>
                    </entry>
                </value>
            </entry>
        </rootObject>'''
    params = {
        'IBIRS_action': 'run',
        'IBIRS_args': turn_off_redirection_xml
    }
    wf_response = wf_sess.get(
        f'{ibi_rest_url}/ibfs/WFC/Repository/Public/{report_name}',
        params=params
    )
    report = wf_response.content
    content_type = wf_response.headers.get('Content-Type')
    if 'text/html' in content_type:
        response = make_response(report)
        return response
    elif 'image' in content_type: # send image as base 64 encoded data url
        report_image = b64encode(report).decode('utf-8')
        report_html = f'''
            <html><body align="middle">
                
            </body></html>'''
        response = make_response(report_html)
        return response
    response = make_response(report)
    response.headers.set('Content-Type', content_type)
    return response

```

Creating an Interface to Display WebFOCUS Reports in Flask App

Through a combination of app.py and run_reports.html we are able to create this iFrame within our Flask application to display a report from WebFOCUS.



In app.py, we have the route run_reports to set up the page in the Flask application.

The function run_reports creates a template within the Flask application to allow for the HTML code to appear in the application.

```
@app.route('/run_reports')
def run_reports():
    files_xml = list_files_in_path_xml(file_type="FexFile")
    # reports is a list of report names
    reports = files_xml_to_list(files_xml)
    return render_template('run_reports.html', reports=reports)
```

You can pass a list of report names for the user to select through a drop-down list. The report will then be run and loaded into an iFrame, as shown in the HTML code in the next section.

Run_reports.html - Creating an iFrame to Display WebFOCUS Report

Within the Flask application, you have created an iFrame in the HTML code for the run_report page that allows for the WebFOCUS report to be displayed. The form within the HTML code makes a request to the Flask application to go to the run_report route and run the report.

The code for run_reports.html is as follows:

```
<form action="/run_report" id="run_report" target="report_frame"
method="post">
  <h2> Report Name: </h2>
  <select name = "report_name" id="report_name">
    {% for rep_name in reports %}
      <option value="{{rep_name}}">{{rep_name}}</option>
    {% endfor %}
  <input type="submit" value = "Submit" id="button"

  />
</form>

<p>Note: Reports will show up in the frame below and may take some time to
load; please wait a moment after clicking run.</p>
<p>Reports assumed to be in the Public Repository and are case sensitive.</p>

<iframe src="about:blank" id="report_frame" name="report_frame"
style="height: 100%; width: 90% ">
</iframe>
```

ReportCaster Schedules

Running a ReportCaster schedule is very similar to running a report, but is in fact much easier. The response will return a '200' status code and an HTML of schedule job ID upon success, or a '404' status code with XML of failure response code if not.

To make using schedules easier, the List Files in the Path function is expanded for ReportCaster schedules to create a schedule table with more detail beyond just the schedule name.

Running a ReportCaster Schedule

Similar to the Run Report section, you can create a `/run_schedule` route to run the schedule.

The Python Flask code to do so is as follows:

```
@app.route('/run_schedule', methods=['POST'])
def run_schedule():
    schedule_name = request.form.get('schedule_name')
    wf_sess = wf_login()
    payload = {
        'IBIRS_action': 'run',
    }
    if wf_sess.IBIWF_SES_AUTH_TOKEN is not None:
        payload['IBIWF_SES_AUTH_TOKEN'] = wf_sess.IBIWF_SES_AUTH_TOKEN

    response = wf_sess.post(
        f'{ibi_rest_url}/ibfs/WFC/Repository/Public/{schedule_name}',
        data=payload
    )

    if response.status_code == 200:
        flash(f"Successfully added schedule: {schedule_name} to the queue.")
    elif response.status_code == 404:
        flash(f"Error: Could not run schedule {schedule_name}")
    else:
        flash(f"Undetermined error; Response status code:{response.status_code}")
    return redirect(request.referrer)
```

A message is flashed to the user depending on the success of running the schedule. For this app, `return redirect(request.referrer)` will take the user back to the schedules page when the schedule is run.

There are two schedule pages: a simple version with a drop-down list of schedule pages such as 'run_reports', and a more detailed table of schedule information, but with the same features.

The request.referrer will take the user back to whichever version they initiated the request with.

Creating a Simple Table of Schedules

You can replicate the /run_reports page into a /schedules page that provides equivalent functionality.

app.py:

```
# Schedules home page: get dropdown of schedules in the Public Repository
@app.route('/schedules')
def schedules():
    sched_files_xml = list_files_in_path_xml(file_type="CasterSchedule")
    schedules = files_xml_to_list(sched_files_xml)
    return render_template('schedules.html', schedules=schedules)
```

schedules.html:

```
<h3>Run or View Details for an existing schedule:</h3>
<form><p> Schedule Name:</p>
  <select name = "schedule_name" id="schedule_name">
    {% for sched_name in schedules %}
      <option value="{{sched_name}}">{{sched_name}}</option>
    {% endfor %}
  </select>
  <input type="submit" value = "Run Schedule"
    formaction="/run_schedule" formmethod="POST" id="button"/>

  <input type="submit" value = "Get Info + Log"
    formaction="/view_schedule_log" formmethod="GET" id="button">
</form>

<form method="GET" >
  <input type="submit" name="expand" id="view_expanded_schedule"
    value="View Expanded Schedule Table">
  </input>
</form>

{% with messages = get_flashed_messages() %}
  {% if messages %}
    {% for message in messages %}
      <p style="color:darkred"><strong> {{ message }} </strong></p>
    {% endfor %}
  {% endif %}
{% endwith %}
```

The jinja2 segment at the end displays the flashed message based on the response of the form submission to run a schedule.

However, you can additionally give users the option to view a full table of contents of the schedules, and add the option of viewing its information and log in a new page. Before creating these features, this is how the page would look when rendered:

<http://localhost:8080/schedules> :

Run or View Details for an existing schedule:

Schedule Name:

AHTML Chart Schedule.sch ▾

Run Schedule

Get Info + Log

View Expanded Schedule Table

When the *View Expanded Schedule Table* button is clicked, it submits a get request with the “expand” key set to *View Expanded Schedule Table*.

You must process this in Flask. The new `/schedules` route will appear as the following.

app.py:

```
@app.route('/schedules')
def schedules():
    sched_files_xml = list_files_in_path_xml(file_type="CasterSchedule")
    if not request.args.get("expand"):
        schedules = files_xml_to_list(sched_files_xml)
        return render_template('schedules.html', schedules=schedules)
    else:
        schedule_items = dict()
        for item in sched_files_xml: # always a schedule item
            item_dict = {}
            item_name = item.get('name')
            item_dict['desc'] = item.get('description')
            item_dict['summary'] = item.get('summary')
            # 13 digit unix epoch time in ms listed in xml
            # Convert to 10 digit secs unixtime then format as datetime string
            unixtime_created_ms = int(item.get('createdOn'))
            datetime_created = unixtime_ms_to_datetime(unixtime_created_ms)
            item_dict['creation_time'] = datetime_created
            # casterObject is a child of item
            casterObject = item.find('casterObject')

            # Only support email schedules
            if casterObject.get('sendMethod') != 'EMAIL':
                continue

            # Get destination address, owner, last time executed by
            # parsing property tagged entries;
            item_dict['destinationAddress'] = casterObject.get(
                'destinationAddress')

            item_dict['owner'] = casterObject.get('owner')
            schedule_items[item_name] = item_dict
        # Creates a list of 2-tuples (item_name, item_dict) sorted by
        # datecreated, most to least recent
        schedule_items_list = sorted(
            schedule_items.items(),
            key=lambda x: x[1]['creation_time'],
            reverse=True
        )
        return render_template(
            'schedules.html',
            schedules=schedule_items_list,
            expand=True
        )
```

You can loop through all the scheduled objects in the XML response and receive all relevant details, storing them into a dictionary. Because the time values WebFOCUS uses are unixtime in ms, you must convert them into readable date/time format. You can use this again in a later section, to create a separate function to handle the following syntax:

```
def unixtime_ms_to_datetime(unixtime_ms):  
    unixtime = unixtime_ms/1000  
    datetime_created = datetime.datetime.fromtimestamp(unixtime)  
    datetime_string = datetime_created.strftime("%Y-%m-%d %H:%M:%S")  
    return datetime_string
```

Currently, only schedules with email destinations are supported.

To expand the functionality, a separate function to parse XML into a formatted list or dictionary should be created, which properly adapts to the destination type. However, this is an exercise in parsing XML in Python, so it will not be included in this document. XML schedule format can be found in the REST WebFOCUS documentation, included in the appendix.

The `schedule_items_list` that is passed to the template is sorted by most to least recent creation time. A list is used instead of a dictionary to preserve the sorted order, and for simplicity. However, there are other possible data structures that are optimal for this usage like an ordered dictionary.

The following images show how the new 'schedules.html' file will appear.

```

{% if not expand %}

<h3>Run or View Details for an existing schedule:</h3>
<form><p> Schedule Name:</p>
  <select name = "schedule_name" id="schedule_name">
    {% for sched_name in schedules %}
      <option value="{{sched_name}}">{{sched_name}}</option>
    {% endfor %}
  </select>
  <input type="submit" value = "Run Schedule"
    formaction="/run_schedule" formmethod="POST" id="button"/>

  <input type="submit" value = "Get Info + Log"
    formaction="/view_schedule_log" formmethod="GET" id="button">
</form>

<form method="GET" >
  <input type="submit" name="expand" id="view_expanded_schedule"
    value="View Expanded Schedule Table">
  </input>
</form>

```

```

{% else %}

<h2>Existing schedules:</h2>

<table align="center" border=1>
  <th>Schedule Name</th>
  <th>Creation Time</th>
  <th>Description</th>
  <th>Summary</th>
  <th>Destination Address</th>
  <th>Owner</th>
  <th>Action</th>

  {% for name, item_dict in schedules %}
    <tr>
      <td>{{name}}</td>
      <td>{{item_dict.creation_time}}</td>
      <td>{{item_dict.desc}}</td>
      <td>{{item_dict.summary}}</td>
      <td>{{item_dict.destinationAddress}}</td>
      <td>{{item_dict.owner}}</td>
      <td>
        <form>
          <input type="hidden" name="schedule_name" value="{{name}}"/>
          <input type="submit" value = "Run Schedule"
            formaction="/run_schedule" formmethod="POST" id="button"/>
          <input type="submit" value = "Get Info + Log"
            formaction="/view_schedule_log" formmethod="GET" id="button">
        </form>
      </td>
    </tr>
  {% endfor %}
</table>
{% endif %}

{% with messages = get_flashed_messages() %}
  {% if messages %}
    {% for message in messages %}
      <p style="color:darkred"><strong> {{ message }} </strong></p>
    {% endfor %}
  {% endif %}
{% endwith %}

```

Note: The table headers are just keys of the dictionary that were created to potentially create a headers list and pass that in to generate the table headers.

Also note that the form actions and options for each table row remains the same from the original drop-down list.

The schedules table at

<http://localhost:5000/schedules?expand=View+Expanded+Schedule+Table> will look like:

Existing schedules:

Schedule Name	Creation Time	Description	Summary	Destination Address	Owner	Action	
AHTML_Chart_Schedule.sch	2019-08-14 10:31:47	AHTML_Chart_Schedule	None	hamza_qureshi@ic.ibi.com	admin	Run Schedule	Get Info + Log
Image_Chart_Schedule.sch	2019-08-14 10:31:12	Image_Chart_Schedule	None	hamza_qureshi@ic.ibi.com	admin	Run Schedule	Get Info + Log
Chart1.sch	2019-08-05 09:39:38	Chart1	None	emily_nesson@ic.ibi.com	admin	Run Schedule	Get Info + Log
TestSchedule.sch	2019-07-22 14:22:53	TestSchedule	Testing WF	hamza_qureshi@ic.ibi.com	admin	Run Schedule	Get Info + Log

Schedules assumed to be in the Public Repository and are case sensitive

Selecting *Run Schedule* and receiving a valid success response from WebFOCUS will give the user the following message:

Existing schedules:

Schedule Name	Creation Time	Description	Summary	Destination Address	Owner	Action	
AHTML_Chart_Schedule.sch	2019-08-14 10:31:47	AHTML_Chart_Schedule	None	hamza_qureshi@ic.ibi.com	admin	Run Schedule	Get Info + Log
Image_Chart_Schedule.sch	2019-08-14 10:31:12	Image_Chart_Schedule	None	hamza_qureshi@ic.ibi.com	admin	Run Schedule	Get Info + Log
Chart1.sch	2019-08-05 09:39:38	Chart1	None	emily_nesson@ic.ibi.com	admin	Run Schedule	Get Info + Log
TestSchedule.sch	2019-07-22 14:22:53	TestSchedule	Testing WF	hamza_qureshi@ic.ibi.com	admin	Run Schedule	Get Info + Log

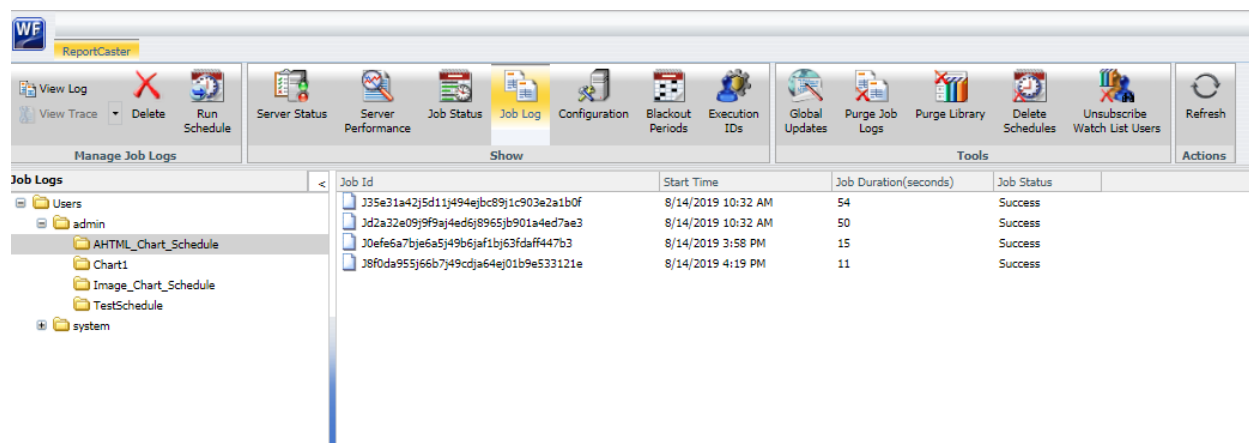
Schedules assumed to be in the Public Repository and are case sensitive

Successfully added schedule: Chart1.sch to the queue.

Selecting *Get Info + Log* will be covered in the following section. It will display a page with tables of more detailed information and logs for the selected schedule.

Get Schedule Info and Log

A schedule ID can be used to retrieve log information about its run history. On WebFOCUS, this can be found on the ReportCaster Status page, as shown in the following image.



The jobs on WebFOCUS can be clicked on, to receive a more detailed description of the job-specific log. For this application, however, you will only provide basic schedule and log information akin to what is shown above. The final table appears, as shown in the following image.

Schedule Information:

Name	AHTML_Chart_Schedule.sch
Owner	admin
ID	S3a8a39b8s192as4db5sa997sb740b2622399
Description	AHTML_Chart_Schedule
Summary	None
Send Method	EMAIL
Destination Address	hamza_qureshi@ic.ibi.com
Last Time Executed	2019-08-14 16:19:44
Status Last Executed	NOERROR
Next Run Time	None Scheduled
Procedures	['IBFS:/WFC/Repository/Public/AHTML_Chart.fex']

Logs:

Start Time	End Time	Error?	Owner
2019-08-14 16:19:44	2019-08-14 16:19:55	None	admin
2019-08-14 15:58:05	2019-08-14 15:58:20	None	admin
2019-08-14 10:32:06	2019-08-14 10:32:56	None	admin
2019-08-14 10:32:00	2019-08-14 10:32:54	None	admin

Procedure: How to Schedule Information and Log Retrieval

1. Get the Schedule name and file path. This will be done using a form from the front end schedule page.
2. Get information about the schedule file from the standard WebFOCUS REST web service. This will contain the information in the first table, most significantly the schedule ID.
3. Use the schedule ID to get log information through the WebFOCUS Log Service REST.
4. Display information in readable format.

Submitting Schedule Name

Getting the schedule name and path is the same as in the other section where variables were used to run it. The first part of the /view_schedule_log route appears as shown in the following image.

```
@app.route('/view_schedule_log', methods=['GET'])
def view_schedule_log():
    schedule_name = request.args.get('schedule_name')
    # If no schedule requested, give users a dropdown of available
    if not schedule_name:
        files_xml = list_files_in_path_xml(file_type="CasterSchedule")
        # schedules is a list of schedule names
        schedules = []
        for item in files_xml:
            schedule_name = item.get("name")
            schedules.append(schedule_name)
    return render_template(
        "schedule_log_info.html", schedule=None, schedules=schedules
    )
```

Note: All Python code in the remainder of this section will be using the /view_schedule_log route.

The associated HTML code will look like:

```
{% if not schedule %}
    <h2>Please select a schedule:</h2>
    <form action="/view_schedule_log" method="GET" >
        <select name = "schedule_name" id="schedule_name">
            {% for sched_name in schedules %}
                <option value="{{sched_name}}">{{sched_name}}</option>
            {% endfor %}
        <input type="submit" value = "Get Info + Log"
            " id="button"
        />
    </form>
{% else %}
```

The following image shows how it will appear.

Please select a schedule:

The image shows a web form with a light blue background. At the top, the text "Please select a schedule:" is displayed in a dark blue, serif font. Below this text is a white dropdown menu with a dark blue border. The dropdown menu is open, showing the text "Chart1.sch" and a small downward-pointing triangle on the right side. To the right of the dropdown menu is a dark blue button with the text "Get Info + Log" in white, sans-serif font.

This is basically the same as the /schedules route with only the option of getting info and log.

The Flask application will pass in the schedules list only if no schedule is selected. Otherwise, it will pass in the schedule name (as "schedule") since only one schedule will be considered on this page.

When a schedule name is passed in, you can split the view schedule log route into two parts:

- Retrieving the schedule information.
- Retrieving the schedule log.

Retrieving Schedule Information

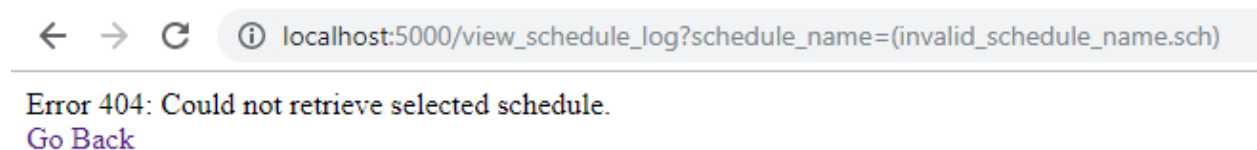
To receive information about a file through WebFOCUS REST web services, you can use the IBIRS_action of get, using the IBFS service. This will return an XML object with the object as a child tag of the rootObject. You will know what type of object it will be based on the extension of the file. In this case, it will be a CasterObject.

To check if the file entered is valid, you must check if the IBFS response object's status code is 10000 which indicates a success. A different response code indicates failure, and will break your XML parsing attempt.

The Python code to do this appears in the following image.

```
wf_sess = wf_login()
# Get schedule xml object
params = {'IBIRS_action': 'get'}
response = wf_sess.get(
    f'{ibi_rest_url}/ibfs/WFC/Repository/Public/{schedule_name}',
    params=params
)
# Parse xml for schedule id
if response.status_code != 200:
    return 'Error: Could not communicate with WebFOCUS Client' + \
        f'<br> <a href="{url_for("schedules")}">Go Back</a>', 404
root = ET.fromstring(response.content)
if root.attrib['returncode'] != "10000":
    print("error retcode != 10k")
    return 'Error 404: Could not retrieve selected schedule.' + \
        f'<br> <a href="{url_for("schedules")}">Go Back</a>', 404
```

If the schedule is not listed, a simple HTML will include the given text and a link to return to the main schedules page:



Otherwise, we will need to parse the XML.

The rootObject will be a child of the IBFS Response object and will contain the schedule ID, which will be needed for the schedule log section.

The casterObject will contain all other relevant information, and will be a child of the rootObject.

From the casterObject XML, in this app, it will parse for:

- Schedule Name
- Owner
- ID
- Description
- Summary
- Send Method
- Destination Address
- Last Time Executed
- Status Last Executed
- Next Run Time
- Procedures

The rest of the given code is an exercise in parsing this XML, as shown in the following image.

```
for child in root:
    if child.tag == 'rootObject':
        rootObject = child
    schedule_id = rootObject.attrib['handle']
    # Parse xml for more schedule information
    for child in rootObject:
        if child.tag == 'casterObject':
            casterObject = child
        lastTimeExecuted_unix = casterObject.get('lastTimeExecuted')
        if lastTimeExecuted_unix:
            lastTimeExecuted = unixtime_ms_to_datetime(int(lastTimeExecuted_unix))
        nextRunTime = casterObject.get('nextRunTime')
        if not nextRunTime:
            nextRunTime = "None Scheduled"
        schedule = {
            'Name': schedule_name,
            'Owner':    casterObject.get('owner'),
            'ID':    schedule_id,
            'Description':    casterObject.get('description'),
            'Summary':    casterObject.get('summary'),
            'Send Method':    casterObject.get('sendMethod'),
            'Destination Address':    casterObject.get('destinationAddress'),
            'Last Time Executed':    lastTimeExecuted,
            'Status Last Executed':    casterObject.get('statusLastExecuted'),
            'Next Run Time':    nextRunTime,
            'Procedures':    []
        }
    # Parse xml for procedure information

    for child in casterObject:
        if child.tag == 'taskList':
            taskList = child

    # taskList can have multiple items
    for item in taskList:
        procedureName = item.get('procedureName')
        schedule['Procedures'].append(procedureName)
```

The schedule information is stored in a simple dictionary because in the web page, a simple two-column table with the key-value pairs for each attribute is provided.

The Jinja2 and HTML code to parse through this dictionary appears in the following image.

```
{# continuing from the no schedule selected HTML code from previous section #}
{% else %}
    <table>
        {% for key,value in schedule.items() %}
            <tr>
                <td><strong>{{key}}</strong></td>
                <td>{{value}} </td>
            </tr>
        {% endfor %}
    </table>
```

The following image shows the rendered web page.

Schedule Information:

Name	Chart1.sch
Owner	admin
ID	S7fc17117s93a4s411csa0ddsf240b61bf143
Description	Chart1
Summary	None
Send Method	EMAIL
Destination Address	emily_nesson@ic.ibi.com
Last Time Executed	2019-08-19 10:33:26
Status Last Executed	NOERROR
Next Run Time	None Scheduled
Procedures	['IBFS:/WFC/Repository/Public/Chart1.fex']

You can now, use the schedule ID to retrieve its logs.

Retrieving Schedule Job Log

The format of WebFOCUS Log REST responses is different from the normal IBFS REST service in that it uses namespaces, which makes parsing a bit more involved. However, the general concepts are the same.

You can make the REST call to the LogServiceREST URL, then parse the XML response and attributes to create readable values to use in our table.

The code to make the call and receive a response is shown in the following image.

```
# Have schedule id, now use it to retrieve log list
url = f"{ibi_client_protocol}://{ibi_client_host}:{ibi_client_port}" + \
    "/ibi_apps/services/LogServiceREST/getLogInfoListByScheduleId"

params = dict()
params['scheduleId'] = schedule_id

log_response = wf_sess.get(url, params=params)
if log_response.status_code != 200:
    error = f"Could not receive log data for {schedule_name}"
    return render_template('schedule_log_info.html', schedule=schedule_id,
                           error=error)
```

If the schedule has no log data response, then you can only display the information collected above. You can add to the HTML code:

```
{% if not log_data %}
    {% if error %}
        <p><strong>Error:</strong> {{ error }}
    {% endif %}
{% endif %}
```

Otherwise, you will parse the log data XML response and display it in a table.

The following image shows an example of a LogServiceREST XML response.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <ns:getLogInfoListByScheduleIdResponse xmlns:ax279="http://dslog.data.api.broker.ibi/xsd" xmlns:ax276="http://io.java/xsd"
  xmlns:ax275="http://rmi.java/xsd" xmlns:ax273="http://schedule.data.api.broker.ibi/xsd" xmlns:ns="http://ws.api.broker.ibi">
  - <ns:return xsi:type="ax279:DsLog" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ax279:IBFSObjectType>0</ax279:IBFSObjectType>
    <ax279:description xsi:nil="true"/>
    <ax279:endTime>2019-08-14T10:32:54.866-04:00</ax279:endTime>
    <ax279:errorType>0</ax279:errorType>
    <ax279:ibfsId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:ibfsId>
    <ax279:ibfsPath/>
    <ax279:id>J35e31a42j5d11j494ejbc89j1c903e2a1b0f</ax279:id>
    <ax279:jobId>J35e31a42j5d11j494ejbc89j1c903e2a1b0f</ax279:jobId>
    <ax279:logElementList xsi:nil="true"/>
    <ax279:name xsi:nil="true"/>
    <ax279:owner>admin</ax279:owner>
    <ax279:scheduleDescription>AHTML_Chart_Schedule</ax279:scheduleDescription>
    <ax279:scheduleId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:scheduleId>
    <ax279:startTime>2019-08-14T10:32:00.099-04:00</ax279:startTime>
    <ax279:summary xsi:nil="true"/>
  </ns:return>
  - <ns:return xsi:type="ax279:DsLog" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ax279:IBFSObjectType>0</ax279:IBFSObjectType>
    <ax279:description xsi:nil="true"/>
    <ax279:endTime>2019-08-14T10:32:56.316-04:00</ax279:endTime>
    <ax279:errorType>0</ax279:errorType>
    <ax279:ibfsId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:ibfsId>
    <ax279:ibfsPath/>
    <ax279:id>Jd2a32e09j9f9aj4ed6j8965jb901a4ed7ae3</ax279:id>
    <ax279:jobId>Jd2a32e09j9f9aj4ed6j8965jb901a4ed7ae3</ax279:jobId>
    <ax279:logElementList xsi:nil="true"/>
    <ax279:name xsi:nil="true"/>
    <ax279:owner>admin</ax279:owner>
    <ax279:scheduleDescription>AHTML_Chart_Schedule</ax279:scheduleDescription>
    <ax279:scheduleId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:scheduleId>
    <ax279:startTime>2019-08-14T10:32:06.197-04:00</ax279:startTime>
    <ax279:summary xsi:nil="true"/>
  </ns:return>
  - <ns:return xsi:type="ax279:DsLog" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <ax279:IBFSObjectType>0</ax279:IBFSObjectType>
```



```

- <ns:return xsi:type="ax279:DsLog" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ax279:IBFSObjectType>0</ax279:IBFSObjectType>
  <ax279:description xsi:nil="true"/>
  <ax279:endTime>2019-08-14T15:58:20.088-04:00</ax279:endTime>
  <ax279:errorType>0</ax279:errorType>
  <ax279:ibfsId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:ibfsId>
  <ax279:ibfsPath/>
  <ax279:id>J0efe6a7bje6a5j49b6jaf1bj63fdaff447b3</ax279:id>
  <ax279:jobId>J0efe6a7bje6a5j49b6jaf1bj63fdaff447b3</ax279:jobId>
  <ax279:logElementList xsi:nil="true"/>
  <ax279:name xsi:nil="true"/>
  <ax279:owner>admin</ax279:owner>
  <ax279:scheduleDescription>AHTML_Chart_Schedule</ax279:scheduleDescription>
  <ax279:scheduleId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:scheduleId>
  <ax279:startTime>2019-08-14T15:58:05.740-04:00</ax279:startTime>
  <ax279:summary xsi:nil="true"/>
</ns:return>
- <ns:return xsi:type="ax279:DsLog" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ax279:IBFSObjectType>0</ax279:IBFSObjectType>
  <ax279:description xsi:nil="true"/>
  <ax279:endTime>2019-08-14T16:19:55.187-04:00</ax279:endTime>
  <ax279:errorType>0</ax279:errorType>
  <ax279:ibfsId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:ibfsId>
  <ax279:ibfsPath/>
  <ax279:id>J8f0da955j66b7j49cdja64ej01b9e533121e</ax279:id>
  <ax279:jobId>J8f0da955j66b7j49cdja64ej01b9e533121e</ax279:jobId>
  <ax279:logElementList xsi:nil="true"/>
  <ax279:name xsi:nil="true"/>
  <ax279:owner>admin</ax279:owner>
  <ax279:scheduleDescription>AHTML_Chart_Schedule</ax279:scheduleDescription>
  <ax279:scheduleId>S3a8a39b8s192as4db5sa997sb740b2622399</ax279:scheduleId>
  <ax279:startTime>2019-08-14T16:19:44.407-04:00</ax279:startTime>
  <ax279:summary xsi:nil="true"/>
</ns:return>
</ns:getLogInfoListByScheduleIdResponse>

```

The Python ET library will by default resolve namespace tags in the form of "{url}tag", so when working with this XML, you will need to split by '}' to get the tag.

Most of the Python code below will deal with formatting time and error types into meaningful values using a string formatter function dictionary. For each job/log item, output the data to a dictionary. This dictionary will be stored in a list of log_item dictionaries, which will be sorted by creation time, to be used in creating the HTML table output.

The code is as follows:

```
log_root = ET.fromstring(log_response.content)
# log_data is a list of log_item attribute dictionaries
log_data = list()
# tags are of the form "{url}tag" in the xml; parse out the actual tag
format_tag = lambda x: x.split('}')[1]
# timestamps are of the form "yyyy-mm-ddThh:mm:ss.xxx-xx:xx;
# omit anything after seconds; split will return tuple as
# (date_string, time_string)
# so join these as a string separated by a space
def format_time(time_string):
    date_str, time_str = time_string.split('T')
    time_str = time_str[:8] # first 8 digits are hh:mm:ss
    return f"{date_str} {time_str}"
# errorType will be a string of a 1-digit code, mapped in this dictionary:
error_code_values = {
    "0": "None",
    '1': 'Error',
    '2': 'Warning',
    '6': 'Running',
    '7': 'Running With Error'
}
format_error = lambda error_code: error_code_values.get(error_code)
# relevant xml data for table column headers as keys.
# text formatter function as value
log_formatter = {
    'startTime': format_time,
    'endTime': format_time,
    'errorType': format_error,
    'owner': lambda x: x, # Nothing to format
}
```

```

for log_item in log_root:
    # log item exists for each time schedule was run
    # attributes is a dictionary of each log item's data
    attributes = dict()
    for attribute in log_item: # attributes are children of xml log items
        if attribute.text: # only care for attributes with text
            key = format_tag(attribute.tag)
            if key in log_formatter:
                # function from log_formatter that will format the xml text
                format_func = log_formatter[key]
                # if key is 'owner', nothing to format
                attributes[key] = format_func(attribute.text)
    log_data.append(attributes)

# sort data by start time, most recent to least recent
log_data.sort(key=lambda x: x['startTime'], reverse=True)

return render_template(
    'schedule_log_info.html',
    schedule=schedule,
    log_data=log_data
)

```

To process this in the HTML, use the following code:

```

{% else %}
    <h2>Logs:</h2>
    <table>
        <th>Start Time</th>
        <th>End Time</th>
        <th>Error?</th>
        <th>Owner</th>

        {% for log_item in log_data %}
            <tr>
                <td>{{log_item.startTime}}</td>
                <td>{{log_item.endTime}}</td>
                <td style=color:{{'limegreen' if log_item.errorType=='None'
else 'orange' if log_item.errorType=='Running' else 'red'}}>
                    <strong>{{log_item.errorType}}</strong>
                </td>
                <td>{{log_item.owner}}</td>
            </tr>
        {% endfor %}
    </table>
{% endif %}
{% endif %}

```

The styling for `errorType` simply adjusts the color between green, orange, and red depending on the status.

The following image shows a sample rendering.

Logs:

Start Time	End Time	Error?	Owner
2019-08-19 12:42:42	2019-08-19 12:42:52	None	admin
2019-08-14 16:19:44	2019-08-14 16:19:55	None	admin
2019-08-14 15:58:05	2019-08-14 15:58:20	None	admin
2019-08-14 10:32:06	2019-08-14 10:32:56	None	admin
2019-08-14 10:32:00	2019-08-14 10:32:54	None	admin

This is all you will do with logs for this sample application. However, there are other Log service REST calls you can make, such as retrieving the full log for a specific job. Python allows you to be creative in how you use and make these calls, such as providing a new link for such information only if there is an error.

Run Report Deferred

This section describes how to run a deferred report from the WebFOCUS public repository.

Requests are made to WebFOCUS to run the report, and the content of the XML that is returned is read to ensure the proper response.

Upon success, you will receive a deferred item ticket that can be used to retrieve the item when it is finished running. Although you will not use this ticket in the application, it may prove useful for associating users with a requested ticket in a real business application.


The ticket will be stored as a file in a folder associated with the user running it. You are signed in as admin in the application, so your tickets will be associated with that user.

Procedure: How to Run a Deferred WebFOCUS Report

1. Make a login connection to WebFOCUS.
2. Add parameters to the payload to request running the report deferred to WebFOCUS.
3. Return the proper response based on the success of the retrieval of the report.

Run a Deferred WebFOCUS Report in the Flask Application

Through a combination of app.py and defer_reports.html you can create this page within the Flask application to display a deferred report from WebFOCUS, with either a message of success or error in its ability to retrieve the requested report.

 [Home](#) [Run Reports](#) [Schedules](#) [View Schedule Log](#) [Defer Reports](#) [View Deferred Reports](#) [Logout](#)

Run a Deferred Report

Process:

Select a report to run deferred. It will return a token that will be saved to your current browser session.

AHTML_Chart.fex ▾

Enter ticket description (required)

Submit

Successfully ran deferred report: AHTML_Chart.fex

View Table of all deferred reports

Defer_reports.html - Creating a Form to Request a Deferred WebFOCUS Report and Outputting Result Messages

Within the Flask application, you have created a form within the HTML code that allows for the drop-down list of report options to choose from, as well as a space to enter a ticket description of the report for retrieval. Also included in defer_reports.html is a series of response messages that are displayed depending on whether the deferred report is ran successfully or not.

The code for defer_reports.html is shown in the following image.

```
<form action="/defer_report" method="post" >
    <select name = "report_name" id="report_name">
        {% for rep_name in reports %}
            <option value="{{rep_name}}">{{rep_name}}</option>
        {% endfor %}
    <br>
    <input type="text" style="width:25%" name = "IBIRS_tDesc"
id="IBIRS_tDesc"
    placeholder = "Enter ticket description (required)" required/>

    <input type="submit" value = "Submit" id="button"/>
</form>
<br>
<br>
{% with messages = get_flashed_messages() %}
    {% if messages %}
        {% for message in messages %}
            <p style="color:darkred"><strong> {{ message }} </strong></p>
        {% endfor %}
    {% endif %}
{% endwith %}
```

Submitting a Request for a Deferred WebFOCUS Report

In app.py, you have the route defer_report to set up the run report deferred request. It works similarly to run_report, but it returns a deferred item ticket rather than the report itself.

The function defer_report simply tells WebFOCUS to run the report deferred, and redirects back to the main defer_reports page with the proper feedback.

```
# Run report deferred and store id info in session
@app.route('/defer_report', methods=['POST'])
def defer_report():
    report_name = request.form.get('report_name')
    tDesc = request.form.get('IBIRS_tDesc')

    wf_sess = wf_login()
    payload = {'IBIRS_action': 'runDeferred' }
    payload['IBIRS_tDesc'] = tDesc
    payload['IBIRS_path'] = f"IBFS:/WFC/Repository/Public/{report_name}"
    payload['IBIRS_parameters'] = "__null"
    payload['IBIRS_args'] = "__null"
    payload['IBIRS_service'] = "ibfs"

    if wf_sess.IBIWF_SES_AUTH_TOKEN is not None:
        payload['IBIWF_SES_AUTH_TOKEN'] = wf_sess.IBIWF_SES_AUTH_TOKEN

    response = wf_sess.post(ibi_rest_url, data=payload)
```

This creates the deferred report request. Next, you must process the response.

Processing Responses for a Deferred WebFOCUS Report

The function `defer_report` reads the response that WebFOCUS gives from the request for a deferred report. The response determines the code that is returned.

For example, code 10000 means that the deferred report was run successfully, as shown in the following image.

```
if response.status_code != 200:
    print("Error status code != 200")
    flash("Error: Could not defer report.")
    return redirect(url_for('defer_reports'))

root = ET.fromstring(response.content)

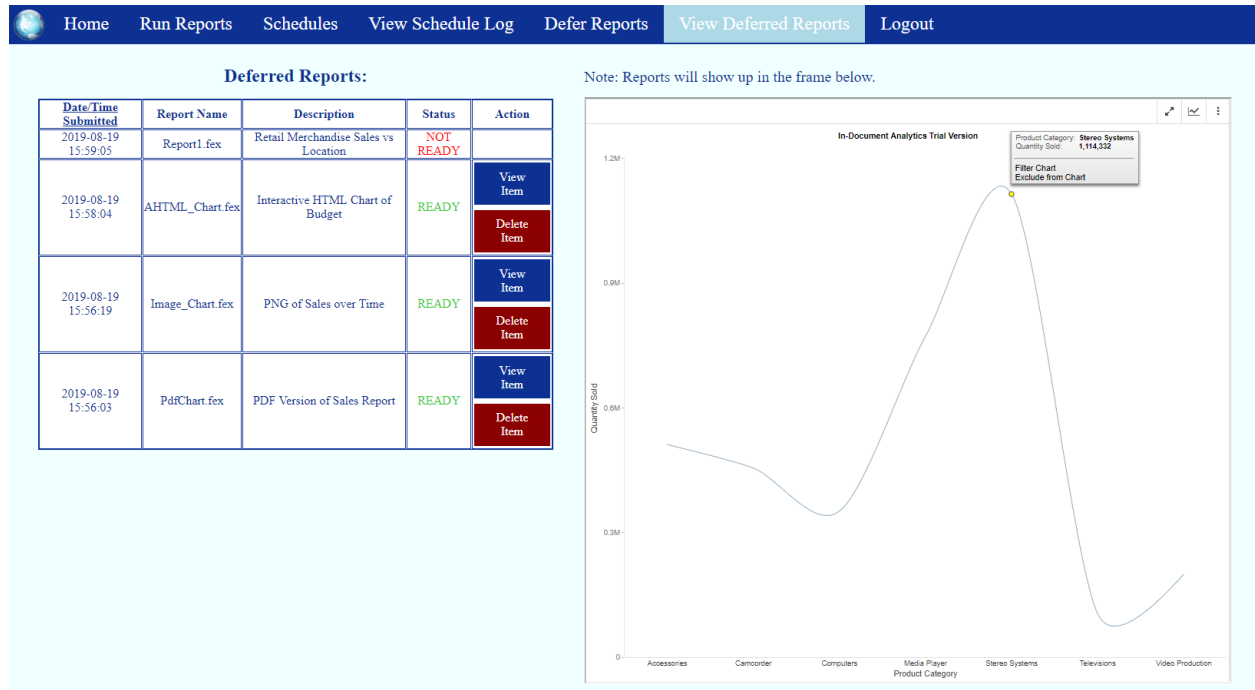
# returncode 10000 means it ran successfully
if root.get('returncode') != "10000":
    print("Error retcode != 10k")
    flash("Error: Could not defer report.")
    return redirect(url_for('defer_reports'))

flash(f"Successfully ran deferred report: {report_name}")
return redirect(url_for('defer_reports'))
```


View and Use Deferred Tickets

In this sample application, you can receive a deferred report from the WebFOCUS public repository and display it using iFrame in the Flask application.

You can create a table to list all deferred items along with their ticket description, date submitted, and status (ready to view or in process). Within the table, you can either view the report in an iFrame or delete it.



App.py - Submitting a Request for a Deferred WebFOCUS Report

In app.py, you have the route deferred_reports_table to set up the table of deferred reports in the Flask application. The function deferred_reports_table is able to sort the table through HTML. It acts to retrieve a list of the deferred tickets and convert the XML response into a python dictionary. This information is then sent to deferred_reports_table.html.

```
@app.route('/deferred_reports_table', methods=['GET'])
def deferred_reports_table():
    if "user_name" not in session:
        return redirect('/')
    wf_sess = wf_login()

    # used to sort table in html
    sort_reversed = True if request.args.get('reverse') == 'True' else False

    # retrieve list of deferred tickets
    payload = {"IBIRS_action": "listTickets"}
    payload['IBIRS_service'] = 'defer'
    payload['IBIRS_filters'] = payload['IBIRS_args'] = '__null'
    payload['IBIWF_SES_AUTH_TOKEN'] = wf_sess.IBIWF_SES_AUTH_TOKEN
    response = wf_sess.get(ibi_rest_url, params=payload) # will be xml
```

Upon receiving the XML response, you can store all information into a dictionary, similar to how you processed schedule information in the ReportCaster section.

```

# convert xml response to minimal dict for easy access
tree = ET.fromstring(response.text)
root = tree.find('rootObject')

deferred_tickets = dict()
for item in root:
    item_dict = {}
    item_name = item.attrib['name']
    item_dict['desc'] = item.attrib['description']
    # 13 digit unix epoch time in ms listed in xml
    # Convert to 10 digit secs unixtime then format as datetime string
    unixtime_created_ms = int(item.attrib['createdOn'])
    item_dict['creation_time'] = unixtime_ms_to_datetime(unixtime_created_ms)
    # status and properties are a child of item in xml
    for node in item:
        # Create sub elements
        if node.tag == 'status':
            status = node.attrib['name']
            item_dict['status'] = 'READY' if status == 'CTH_DEFER_READY' \
                else 'NOT READY'

        # parse property tagged entries; reportname is an attribute
        if node.tag == 'properties':
            for property_node in node:
                if property_node.attrib['key'] == 'IBIMR_fex_name':
                    item_dict['report_name'] = property_node.attrib['value']

    deferred_tickets[item_name] = item_dict

# Creates a list of 2-tuples (item_name, item_dict) sorted by datecreated
# Default is most to least recent; can be changed by reverse flag in query
deferred_tickets = sorted(
    deferred_tickets.items(),
    key=lambda x: x[1]['creation_time'],
    reverse=not sort_reversed
)

return render_template(
    "deferred_reports_table.html",
    deferred_items=deferred_tickets,
    reverse=sort_reversed
)

```

The variable `sort_reversed` is included as an example of how to format data. This can be applied to the other scheduled tables that were created as well. If the user passes in a parameter to sort reversed, or based on a column, it is trivial to use the key to sort using the `sorted` function.

In this application, users are allowed to select this sort option of most to least recent or vice versa by clicking on the date/time column heading, as you will see in the HTML.

Deferred_reports_table.html - Creating a Table of Deferred Reports

Within the Flask application, you have created a table within the HTML code that displays the name of the deferred report, its date/time submitted, its ticket description, its status (ready or still processing), and the ability to view or delete the report.

The code for the table in `defer_reports.html` is shown in the following image.

```

<table>
  <th>
    <a id="x" href = "?reverse={{not reverse}}"><u>Date/Time Submitted</u></a>
  </th>
  <th>Report Name</th>
  <th>Description</th>
  <th>Status</th>
  <th>Action</th>

  {% for name, item_dict in deferred_items %}

    <tr>
      <td>{{item_dict.creation_time}}</td>
      <td>{{item_dict.report_name}}</td>
      <td>{{item_dict.desc}}</td>
      <td style="color:{{'limegreen' if item_dict.status=='READY' else 'red'}}">
        {{item_dict.status}}
      </td>
      <td>
        {% if item_dict.status=='READY' %}
          <form method="post" action="get_deferred_report" target="report_frame"/>
            <button type="submit" name="ticket_name" value="{{name}}" />
              View Item
            </button>
          </form>
          <form method="post" action="delete_item">
            <input type="hidden" name="item_type" value="deferred"/>
            <button type="submit" name="item_name" value="{{name}}" />
              Delete Item
            </button>
          </form>
        {% endif %}
      </td>
    </tr>

  {% endfor %}

</table>

```

Deferred_reports_table.html - Creating a View Item Form

Within the Flask application, you have created a form within the HTML code that allows for the user to view a report in the iFrame from the table.

The code for the form for View Item is shown in the following image.

```
<form method="post" action="get_deferred_report" target="report_frame">  
    <button type="submit" name="ticket_name" value="{{name}}" />  
    View Item  
</button>  
</form>
```

The code that creates the iFrame in which the selected item can be viewed, is shown in the following image.

```
<iframe name="report_frame"  
style="position: absolute; height: 80%; width: 50% " onload="hide()" />  
    Report Appears Here  
</iframe>
```

App.py - Getting a Deferred Report

In app.py, you have the route `get_deferred_report` to retrieve the deferred report data.

The function retrieves the ticket name and, if found, is able to properly respond by displaying the selected item. By clicking the *View Item* button in the table on the Flask application, you can view the selected item in iFrame.

```
# Retrieves deferred report data
@app.route('/get_deferred_report', methods=['GET', 'POST'])
def get_deferred_report():
    ticket_name = request.form.get('ticket_name')
    if not ticket_name:
        return "Error: No ticket selected"
    wf_sess = wf_login()
    params = {
        'IBIRS_action': 'getReport',
        'IBIRS_service': 'defer'
    }
    params['IBIRS_ticketName'] = ticket_name
    response = wf_sess.get(ibi_rest_url, params=params)
    return response.content
```

You should check the response content type headers and proceed with the same steps as `run_report`. However, since you are repeating the same process, a better design would be to create a function to do so.

The other option in the deferred reports table is to delete a report, but since the delete method in general is enough to include other types of WebFOCUS objects such as reports and schedules, you can make a new route that encompasses all possible object types in the following section.

Delete Item

The delete item option in this Flask application is included in the View Deferred Reports section only, but would be similarly applied to any of the other options for viewing different types of reports. The delete item is displayed as a button with form action `/delete_item` and with a hidden input of `item_type =deferred`. The Flask route will delete an item based on its file type and return a proper feedback message.

Deferred Reports:				
Date/Time Submitted	Report Name	Description	Status	Action
2019-08-13 14:27:56	Chart1.fex	Webfocus	READY	<div>View Item</div> <div>Delete Item</div>

App.py - Delete Item

In `app.py`, you have the route `delete_item` to allow for an item to be deleted, as in `deferred_reports_table`.

The function `delete_item` makes a post request to WebFOCUS which uses the `item_name` and file path to delete it from WebFOCUS. You can use `item_type` to make the proper call. However, `deferred` has a separate call to delete tickets. Upon getting those parameters, you are able to delete the selected item.


```

@app.route('/delete_item', methods=['POST'])
def delete_item():
    item_name = request.form.get('item_name')
    item_type = request.form.get('item_type')
    wf_sess = wf_login()

    payload = dict()
    if wf_sess.IBIWF_SES_AUTH_TOKEN is not None:
        payload['IBIWF_SES_AUTH_TOKEN'] = wf_sess.IBIWF_SES_AUTH_TOKEN

    if item_type == 'deferred':
        payload['IBIRS_action'] = 'deleteTicket'
        payload['IBIRS_service'] = 'defer'
        payload['IBIRS_ticketName'] = item_name
        response = wf_sess.post(ibi_rest_url, data=payload)
    else:
        payload['IBIRS_action'] = 'delete'
        response = wf_sess.post(
            f'{ibi_rest_url}/ibfs/WFC/Repository/Public/{item_name}',
            data=payload
        )
    message = f"Deleted Item: {item_name}" if response.status_code == 200 \
        else "Could not delete item"
    flash(message)
    return redirect(request.referrer)

```

Deferred_reports_table.html - Delete Item in Deferred Reports

The code for the delete item portion in deferred_reports_table.html is shown in the following image.

```

<form method="post" action="delete_item" class="inline" onsubmit="load()">
    <input type="hidden" name="item_type" value="deferred"/>
    <button type="submit" name="item_name" value="{{name}}">
        Delete Item
    </button>
</form>

```

The item_name and item_type get processed by the Flask application to delete the deferred ticket.

Appendix: WebFOCUS Report Static CSS/JS Files

By default, WebFOCUS HTML Reports include relative paths for static CSS and JS files used in the HTML files. As a result, when your reports are loaded into iFrame, it will call `http://localhost:5000/ibi_apps/` to access these files.

A simple fix is to create a route in the Flask directory where the `/ibi_apps/<path:path>` route will retrieve the requested `file_path` from WebFOCUS itself.

This is shown in the following code:

```
# Used to receive webfocus report local files (js/css) from proper source
@app.route('/ibi_apps/<path:page>', methods=['GET', 'POST'])
def client_app_redirect(page):
    # Security: Only allow this url to serve content
    # when referred from this application
    referrer_url = request.referrer
    referrer = urllib.parse.urlparse(referrer_url)
    referrer_base_url = f'{referrer.scheme}://{referrer.hostname}'
    referrer_base_url += f':{referrer.port}/' if referrer.port else '/'
    if request.host_url != referrer_base_url:
        abort(403)

    # Use this line of code if you copy ibi html folder into static:
    # return send_from_directory('static', f'ibi_static/{page}')

    base=f'{ibi_client_protocol}://{ibi_client_host}:{ibi_client_port}/ibi_apps/'
    wf_sess = wf_login()

    # Python requests automatically decodes a gzip-encoded response
    # so set stream=True for raw bytes
    response = wf_sess.get(base + page, stream=True)
    # Forward response content and headers to user
    return response.raw.read(), response.status_code, response.headers.items()
```

This can be slow since it must authenticate with WebFOCUS for every file.

While there are several optimizations possible, this will work to receive the static files from WebFOCUS even if the files are inaccessible from the client browser and not on the Python application's machine.

If allowed, storing the static files from the 'ibi/WebFOCUS82/webapps/webfocus' directory on to your server and returning the content via the commented line of code is a much better approach:

```
return send_from_directory('static', f'ibi_static/{page}')
```

Please check the technical library for more details on how to properly use this folder; this can be done via a production server more easily than in Flask.