Automating the Process of Document Submission in the IVOA Document Repository

Kalyani Pedamkar[1], Giulia Iafrate[1], Claudio Vuerli[1], and Marco Molinaro[1]

[1]INAF - Osservatorio Astronomico di Trieste

December 9, 2024

Summary

This report outlines the development of a web application aimed at automating the management of the International Virtual Observatory Alliance (IVOA) Document Repository [1]. The manual process of document submission, metadata management, and page creation has been replaced with an automated system that improves efficiency and accuracy. The application is built using Python Flask, SQLite3, and SQLAlchemy, with additional tools like BeautifulSoup for web scraping to populate the database with existing metadata. It supports structured storage of uploaded files and real-time updates to landing pages categorized by document type, enhancing organization and accessibility. Future enhancements include adding features such as an RFC phase, errata management, and advanced search capabilities. By automating workflows and integrating historical metadata, this system provides a reliable and scalable solution for the IVOA community's document management needs, which otherwise are maintained with laborious direct manual interventions today.

Abstract

The International Virtual Observatory Alliance (IVOA)[2] produces standards to enable the interoperability of astronomical data and services for the global astrophysical community. The working groups and the interest groups submit the documents following the promotion process described by the IVOA document for Standards and Processes. This process is currently managed mostly manually by the IVOA Document Coordinator. We are developing a web application to automate this process. This tool will help users submit the document alongside their metadata information, which will then be stored in the IVOA database. The application will use the metadata to dynamically generate both the individual document landing pages and the global repository landing page. Other features of this application will be the addition of Errata, Requests for Comments (RFC) phases, and other related features, which are currently manually managed by the IVOA Document Coordinator or other IVOA organization's roles. This web application is built using the Python Flask framework, SQLite3 as the database, SQLAlchemy as the Object Relational Mapper (ORM), and various front-end tools. The application is designed to accept metadata and different formats of documents through a form interface. The uploaded documents are then processed, renamed, and saved in a structured document tree. The accompanying metadata is stored in SQLite3 database and used to dynamically generate individual document landing pages for ease of access and organization. This document provides a step-by-step explanation of the architecture, technology and implementation involved in the development of this application.

[1] https://www.ivoa.net/documents/
[2] https://ivoa.net/

Contents

1 Introduction

1.1 Background

In the current system, the upload of documents and their associated metadata is performed manually. The process is both time-intensive and prone to errors, as it requires meticulous effort from the document coordinator. Additionally, the creation and maintenance of individual document landing pages and the global repository page are handled manually. These tasks demand significant resources and coordination, making the system inefficient and difficult to scale as the volume of documents increases. The document preparation process can also be managed by the "ivoatex" tool [3] that provides a partial automation of the packaging needed while loading the documents to the repository. However, it does not simplify the IVOA Document Coordinator's task of updating the repository. Furthermore, there is a pressing need to systematically archive older documents and their metadata alongside newer submissions to ensure comprehensive and organized access to historical and current records. The limitations of this manual approach highlight the necessity for an automated and robust solution that can streamline these processes, enhance accuracy, and reduce administrative overhead.

1.2 Objective

The primary objective of this project is to develop an efficient and automated system for document submission and metadata management. To achieve this, a web application has been created using Python and the Flask web framework. This application uses SQLite as its database for lightweight, file-based storage and SQLAlchemy as the Object Relational Mapper (ORM) to facilitate seamless database interactions. Additionally, various frontend tools have been integrated to enhance the application's usability, providing a user-friendly interface for document coordinators and contributors. This system aims to streamline the submission process, reduce manual overhead, and ensure accurate and organized storage of both historical and new records.

2 System Architecture

2.1 Overview

The application consists of:

- **Frontend Interface:** The front end of the application is designed to ensure a smooth user experience for submitting metadata and managing document records. It incorporates the following technologies:

2.1.1 HTML and CSS:

Used to create the structure and styling of the web application, including the dynamically generated document landing pages and metadata display.

[3] https://github.com/ivoa-std/ivoatex

2.1.2 JavaScript and jQuery:

Enhance interactivity by enabling dynamic content updates, form validation, and seamless navigation between pages. jQuery simplifies AJAX calls for real-time updates without requiring full page reloads.

2.1.3 Flask (WTForms):

Facilitates the creation of robust forms for metadata input, ensuring compliance with IVOA document standards while simplifying user interactions. These forms handle data validation and ensure secure submission of metadata to the backend database.

2.1.4 Jinja Templates:

Dynamically render web pages, integrating metadata and document information fetched from the database into the frontend. This ensures that the landing pages for uploaded documents are always up to date and reflect the latest data.

By combining these tools, the application provides an efficient interface for metadata entry, document organization, and seamless navigation through the repository.

- **Backend Logic:** The back end of the web application is implemented using Python with the Flask framework, providing a robust and modular architecture for managing core functionalities. The primary components of the backend logic are as follows:

2.1.5 Routing and Request Handling:

Flask's routing system is used to map specific URLs to corresponding python functions, enabling clear separation of tasks. For instance, routes handle actions such as displaying the metadata submission form, processing uploaded documents, and dynamically rendering the document repository pages.
Current available routes:
`/new_doc`: Handles metadata input.
`/uploadfile`: Facilitates upload of the tar or zip files.
`/documents`: Fetches metadata and displays a list of all available documents on the Main Document landing page.

2.1.6 Database Management with SQLite and SQLAlchemy:

The application uses SQLite, a lightweight, file-based relational database, to store document metadata. SQLite is ideal for this application due to its simplicity and ease of integration with Flask. SQLAlchemy, an Object Relational Mapper (ORM), is employed to interact with the database through Python objects.
The database schema includes 4 tables:

- **IVOA Table:** This table stores metadata attributes (e.g., Title, Concise_name, docname, status, date, and many more).
- **DOI & Bibcode Table:** This table stores the bibliographic information and the DOI received from the ADS website of the documents that Recommendation or Endorsed Notes.
- **RFC Table:** This table store the link for "Request for Comments" links for the documents that has reached the Proposed Recommendations or Proposed Endorsed Notes.
- **Errata Table:** This table contains the list of Errata links, if any, for a document which has reached a Recommendation stage.

SQLAlchemy models allow for the seamless execution of CRUD (Create, Read, Update, Delete) operations, simplifying data manipulation and retrieval.

2.1.7 File Management:

The file management system is a crucial component of this web application, designed to process and organize uploaded document packages efficiently. The process begins with users uploading compressed files in .tar or .zip formats, which typically contain a collection of files such as PDFs, HTML documents, and LaTeX source files.

2.1.8 File Upload Handling:

Uploaded .tar and .zip files are temporarily stored on the server in a folder named 'Uploads'. Flask's file-handling utilities, along with Python's standard libraries such as 'tarfile' and 'zipfile', are used to identify and extract the contents of these packages.

- **For .tar Files:** .tar files are extracted using the tarfile module. The extraction process iterates through the archived contents, ensuring that all files and folders are unpacked correctly.
- **For .zip Files:** .zip files are handled using the zipfile module, extracting all enclosed files while maintaining the original directory structure.

2.1.9 Renaming and Organizing Extracted Files:

After extraction, the output folder is renamed based on the IVOA Document Standards for naming the documents, which is defined as

`Status-Concise_name-Version_Major.Version_minor-date`

All the attributes for the above are obtained from the metadata provided at the time of form submission. This renamed folder is then stored under a structured path adhering to the IVOA naming conventions within the /documents directory on the web server. This structured storage ensures quick retrieval and alignment

with IVOA standards for document management. The directory hierarchy is as follows:

`/documents/concise_name/date/renamed_folder`

2.1.10 Renaming Individual Files:

The files within the extracted folder (e.g., .pdf, .html) are renamed as per the IVOA Document Standards which is given as

`Status-Concise_name-Version_Major.Version_minor-date.extension`

This ensures consistency and easier identification of files for retrieval.

2.1.11 Storage in the Document Tree Structure:

Once renamed, the folder and its contents are stored in the structured /documents directory. This web application not only facilitates easy retrieval but also ensures compatibility with the web application's dynamic page generation system, allowing users to browse documents based on metadata.

2.1.12 Web Page Generation:

The backend dynamically generates web pages using Flask's Jinja2 templating engine. Metadata and document information fetched from the SQLite database are integrated into HTML templates, allowing users to view up-to-date content for the Individual document landing pages and main repository landing page.

2.2 System Workflow

The application workflow is designed to automate the process of metadata submission, file management, and dynamic web page generation. It consists of the following key steps:

- **Form Submission:** Users interact with the application via a web-based form where they provide document metadata (e.g., title, Concise_name, author, date) and upload a compressed file in .zip or .tar format (Fig 1). The form is implemented using Flask-WTF, which ensures secure and validated data input. The uploaded file and metadata are sent to the backend for further processing.
- **File Processing:**Upon submission, the backend processes the uploaded .zip or .tar file to extract its contents and organize them into a structured directory. The file is untarred or unzipped using Python's tarfile or zipfile modules, respectively. The extracted files (e.g., .pdf, .html) are renamed based on IVOA Document Standards. The processed files are stored in a predefined directory hierarchy under /documents, structured as: /documents/concise_name/date/renamed_folder. This step ensures that the files are properly organized and easily retrievable.

Figure 1: Document Submission Form

- **Database Storage:**The metadata provided by the user is saved in a SQLite database using SQLAlchemy as the ORM. SQLAlchemy provides an abstraction layer, making it straightforward to execute CRUD operations and manage relationships between tables. This centralized metadata storage enables efficient querying and supports the dynamic rendering of web pages.

- **Dynamic Rendering:**The application dynamically generates web pages to present the uploaded documents and their metadata using Flask and Jinja2 templates.

- **Individual Document Landing Pages:** Each document is assigned a dedicated landing page that displays its metadata and provides links to the renamed files (e.g., PDF, HTML, or LaTeX). The content is fetched from the database and rendered dynamically into HTML(Fig 2).


IVOA Documents

ivoa.net

SAMP-Simple Application Messaging Protocol Version 1.3

IVOA REC 2012-04-11

Interest/Working Group:
Applications

Author(s):
M. Taylor, T. Boch, M. Fitzpatrick, A. Allan, J. Fay, L. Paioro, J. Taylor, D. Tody

Editor(s):
T. Boch, M. Fitzpatrick, M. Taylor

DOI:
10.5479/ADS/bib/2012ivoa.spec.1104T

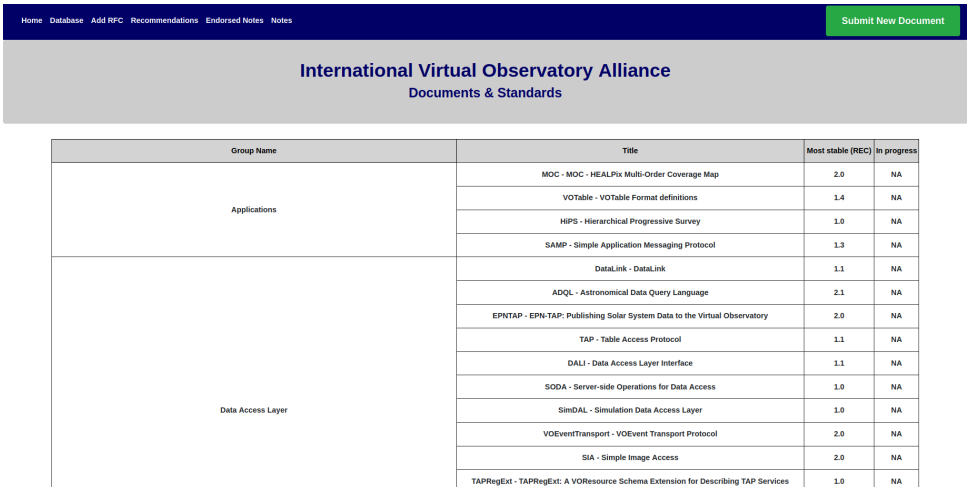
Abstract

SAMP is a messaging protocol that enables astronomy software tools to interoperate and communicate. IVOA members have recognised that building a monolithic tool that attempts to fulfil all the requirements of all users is impractical, and it is a better use of our limited resources to enable individual tools to work together better. One element of this is defining common file formats for the exchange of data between different applications. Another important component is a messaging system that enables the applications to share data and take advantage of each other's functionality. SAMP supports communication between applications on the desktop and in web browsers, and is also intended to form a framework for more general messaging requirements.

Status of the Document

This document has been produced by the Applications Interest Group.
It has been reviewed by IVOA Members and other interested parties, and has been endorsed by the IVOA Executive Committee as an IVOA Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. IVOA's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability inside the Astronomical Community.

Figure 2: Individual Document Landing Page


Home Database Add RFC Recommendations Endorsed Notes Notes

Submit New Document

International Virtual Observatory Alliance

Documents & Standards

Group Name	Title	Most stable (REC)	In progress
Applications	MOC - MOC - HEALPix Multi-Order Coverage Map	2.0	NA
	VOTable - VOTable Format definitions	1.4	NA
	HiPS - Hierarchical Progressive Survey	1.0	NA
	SAMP - Simple Application Messaging Protocol	1.3	NA
Data Access Layer	DataLink - DataLink	1.1	NA
	ADQL - Astronomical Data Query Language	2.1	NA
	EPNTAP - EPN-TAP: Publishing Solar System Data to the Virtual Observatory	2.0	NA
	TAP - Table Access Protocol	1.1	NA
	DALI - Data Access Layer Interface	1.1	NA
	SODA - Server-side Operations for Data Access	1.0	NA
	SimDAL - Simulation Data Access Layer	1.0	NA
	VOEventTransport - VOEvent Transport Protocol	2.0	NA
	SIA - Simple Image Access	2.0	NA
	TAPRegExt - TAPRegExt: A VOResource Schema Extension for Describing TAP Services	1.0	NA

Figure 3: Main Repository Landing Page

- **Main Repository Landing Pages:** The repository landing page ("/documents") serves as the central repository, listing all uploaded documents grouped by the names of the Working groups or Interest groups and their "Most stable" document and the documents "in progress". It provides quick access to individual document landing pages, which contain detailed metadata and links to associated files (Fig 3).

- **Recommendations Page (/rec):** The /rec landing page is dedicated to the list of all recommendations. This classification enables users to easily locate finalized, approved documents that serve as official guidelines or standards.

- **Endorsed Notes Page(/endorsed_notes):** The /endorsed_notes page lists all documents marked as endorsed notes. These documents has been endorsed by the Technical Coordination Group. Endorsed Notes constitute valuable information for the IVOA community, by themselves or as complement of Recommendations.

- **Notes Page (/notes):** The /notes landing page contains a comprehensive list of

```
import os
import flask
import pathlib
from forms import InfoForm, ErrataForm, MoreInfo, RFCForm, DelForm
from flask import (Flask, render_template, url_for, redirect,
                   flash, request, abort, send_from_directory, send_file,
                   current_app, session, json)
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from werkzeug.utils import secure_filename
from werkzeug import FileStorage
from datetime import datetime
from flask_bootstrap import Bootstrap
from sqlalchemy import desc
import zipfile
from tarfile import TarFile
from werkzeug.datastructures import MultiDict
```

Figure 4: Flask Application Installation

all notes that has not yet been endorsed. This page helps users locate exploratory or discussion-oriented documents.

3 Implementation

3.1 Setting Up the Flask Application:

Installation: Install Flask, SQLAlchemy, and import other dependencies (Fig 4)

3.2 Directory Structure:

```
/docrepo/
    ivoa_doc.py
    forms.py
    templates/
    static/
    uploads/
    documents/
```

3.3 Defining the Database Model:

Using SQLAlchemy, a model for metadata is defined with attributes such as title, concise_name, author, date, versions, and more (Fig 5)

3.4 Setting Up Routes and Views:

- **Form Page:** A route renders the HTML form for metadata and file upload.
- **File Upload Handling:** Code to save uploaded the package to a specific path and insert metadata in the database.

```python
db = SQLAlchemy(app)

class Ivoa(db.Model):
    docname =str()

    __tablename__ = 'IVOA'

    id = db.Column(db.Integer)
    group_name = db.Column(db.Text)
    title = db.Column(db.Text)
    concise_name = db.Column(db.Text)
    version_major = db.Column(db.Integer)
    version_minor = db.Column(db.Integer)
    status = db.Column(db.Text)
    date = db.Column(db.Integer, default=datetime.utcnow)
    authors = db.Column(db.Text)
    editors = db.Column(db.Text)
    abstract = db.Column(db.Text)
    docname = db.Column(db.Text, primary_key=True, unique=True)
    package_path = db.Column(db.Text)
    email = db.Column(db.Text)
    comment = db.Column(db.Text)
    contribute = db.Column(db.Text)
    ### One to one relationship between Ivoa and DOI, Bibcode and RFC Link
    doi_bibcode = db.relationship('DOI_Bibcode', backref='ivoa', uselist=False)
    rfc_link = db.relationship('RFC_link', backref='ivoa', uselist=False)
    ### One to many relationship between Ivoa and erratas
    errata = db.relationship('Errata',backref='ivoa',lazy='dynamic')
```

Figure 5: Database Schema

- **Dynamic Page Rendering:** Metadata and files are displayed on a dynamically generated web page.

3.5 Handling File Uploads:

Uploaded .zip or .tar are renamed based on metadata and saved in a structured document tree. (Fig 6)

3.6 Saving Metadata:

The metadata is saved in the SQLite3 database, and SQLAlchemy commits the changes.

3.7 Rendering Metadata on a Web Page:

Metadata is fetched from the database and displayed on a dynamically generated HTML page using Jinja2 templating.

3.8 Populating the Database with Existing Metadata:

After creating the SQLite3 database, it was necessary to populate it with metadata from the existing IVOA Document Repository to ensure continuity and usability. This was achieved by developing a Python script that utilizes web scraping techniques to extract metadata directly from the repository's web pages.

```python
# Upload the Document

@app.route('/uploadfile', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['file']
        filename = secure_filename(file.filename)
        if filename != '':
        #file_ext gives the the extensions of the uploaded file
            file_ext = os.path.splitext(filename)[1]
        # fname is a temparory variable to get the original filename of the the uploaded file using Pathlib's 'stem' function
            fname = pathlib.Path(filename)
            original_filename = fname.stem
            if file_ext not in app.config['UPLOAD_EXTENSIONS']:
                flash('Please upload a .zip or .tar file')
                return redirect(url_for('upload_file'))
                #abort(400)
            else:
        #file.save saves the uploaded file in the 'UPLOAD_DIR' with the original name of the uploaded file
                file.save(os.path.join(app.config['UPLOAD_DIR'], filename))
                source=UPLOAD_DIR+"/"+filename
                destination=UPLOAD_DIR+"/"+Ivoa.docname+file_ext
                os.rename(source,destination)
        #In the step above the saved '.zip' or '.tar' file is renamed

        # Below the '.zip' or '.tar' files are separated and extracted into the 'documents' directory and the folder is renamed as per IVOA standards
        # After renaming the folder, its contents (files inside) are renamed keeping their extensions constant.
                if file_ext == '.zip':
                    with zipfile.ZipFile(destination, 'r') as zip_ref:
                        zip_ref.extractall(path='/var/www/html/docrepo/documents')
                        src= '/var/www/html/docrepo/documents/'+original_filename
                        dst= '/var/www/html/docrepo/documents/'+Ivoa.docname
                        os.rename(src,dst)
                        for name in os.listdir(dst):
                            extension = os.path.splitext(name)[1]
                            new_dst = dst + '/' + Ivoa.docname + extension
                            new_src = dst + '/' + name
                            os.rename(new_src, new_dst)
                elif file_ext == '.tar':
                    with tarfile.open(destination, 'r') as tar_ref:
                        tar_ref.extractall(path='/var/www/html/docrepo/documents')
                        src= '/var/www/html/docrepo/documents/'+original_filename
                        dst= '/var/www/html/docrepo/documents/'+Ivoa.docname
                        os.rename(src,dst)
                        for name in os.listdir(dst):
                            extension = os.path.splitext(name)[1]
                            new_dst = dst + '/' + Ivoa.docname + extension
                            new_src = dst + '/' + name
                            os.rename(new_src, new_dst)
                else:
                    return redirect(url_for('upload_file'))
                return redirect(url_for('thank_you'))
    return render_template('upload.html')
```

Figure 6: File Upload and Storage Process

3.8.1 Tools and Libraries:

The following Python libraries were used for web scraping:

- **BeautifulSoup:** For parsing HTML and extracting metadata from the repository's web pages.
- **Requests:** To send HTTP GET requests and retrieve the content of web pages.
- **SQLite3 Module:** For inserting the extracted metadata directly into the database.

3.8.2 Methodology:

- **Web Page Analysis:** The structure of the IVOA Document Repository web pages was analyzed to identify the location of metadata fields such as document titles, concise names, version numbers, statuses, and dates.
- **Data Extraction:** A Python script was written to parse the HTML content using BeautifulSoup to locate metadata fields within specific HTML tags or attributes (Fig 7).
- **Data Insertion:** The extracted metadata was inserted into the SQLite3 database using SQLAlchemy for ORM-based insertion or the SQLite3 module for direct queries. This ensured that all old and existing metadata records were properly structured and accessible within the new database.

```python
import requests
from datetime import datetime
from bs4 import BeautifulSoup
import re
import sqlite3
#######################################################
#### The URL to be scraped ############################
#######################################################
link = "https://www.ivoa.net/documents/IVOAArchitecture/20211101/index.html"

page = requests.get(link)
soup = BeautifulSoup(page.content, 'html.parser')
results = soup.find()
#######################################################
###### Scrape Concise Name of the Documents ##########
#######################################################
link_parts = link.split('/')

documents_index = link_parts.index('documents')
if documents_index < len(link_parts) - 1 and link_parts[documents_index + 1] == 'cover':
    next_word_index = documents_index + 2
else:
    next_word_index = documents_index + 1

if next_word_index < len(link_parts):
    concise_name = link_parts[next_word_index].split('-')[0]  # Remove everything after '-'
    print("concise_name:", concise_name)
else:
    print("No concise_name found")
```

Figure 7: Snippet of the web scraping code

3.8.3 Outcome:

The web scraping process successfully populated the SQLite3 database with existing metadata from the IVOA repository. This step ensured that the new application would have access to all historical records, maintaining consistency and continuity for users.

4 Deployment

The web application is deployed on an Apache HTTP server, leveraging its robustness and wide adoption for hosting web applications. The deployment process involves the following key steps and configurations:

4.1 Apache Server Setup:

Apache is installed and configured on the hosting environment (e.g., Linux server) to serve the Flask web application. The server handles incoming HTTP requests and routes them to the application.

4.2 WSGI Integration:

Flask applications require a WSGI (Web Server Gateway Interface) middleware to interface with the Apache server. The 'mod_wsgi' module is used to run the Flask application within the Apache environment.

WSGI Script: A WSGI entry-point script is created to define the application callable

```
#!/usr/bin/env python3

import sys
import site

site.addsitedir('/var/www/html/docrepo/virtualenv/lib/python3.6/site-packages')

sys.path.insert(0, '/var/www/html/docrepo')

from ivoa_doc import app as application
```

Figure 8: WSGI Script

```
##

<VirtualHost *:80>
    ServerName www.testingdocrepo.ivoa.info
    ServerAlias testingdocrepo.ivoa.info
    ServerAdmin ivoadoc@ivoa.net

    WSGIDaemonProcess docrepo user=apache group=apache threads=5
#    WSGIDaemonProcess docrepo python-path=/var/www/html/docrepo/venv/lib/python3.6/site-packages/
    WSGIScriptAlias / /var/www/html/docrepo/doc.wsgi

    DocumentRoot "/var/www/html/docrepo/"

    <Directory /var/www/html/docrepo/>
      WSGIProcessGroup docrepo
      WSGIApplicationGroup %{GLOBAL}
      Require all granted
    </Directory>

   ErrorLog "logs/testingdocrepo_error.log"
   CustomLog "logs/testingdocrepo_access.log" combined

</VirtualHost>
```

Figure 9: Virtual Host Configuration

and link it with the Apache server. This script enables Apache to communicate with the Flask application seamlessly(Fig 8).

4.3 Virtual Host Configuration:

A virtual host configuration is created in Apache to define the specific settings for hosting the web application (Fig 9). This configuration file typically includes:

- The domain or subdomain for accessing the application.
- Paths to the WSGI script and application directory.
- Logging configurations for monitoring errors and access.

4.4 File Permissions:

The application directory and its contents are configured with appropriate file and folder permissions to ensure security and accessibility by the Apache user.

4.5 Log Monitoring:

Apache logs are utilized to monitor the application's performance and errors:

- **Error Log:** Captures application-level and server-level errors.
  Location: `/var/log/httpd/testingdocrepo_error.log`
- **Access Log:** Records all incoming requests for analysis and debugging.
  Location: `/var/log/httpd/testingdocrepo_access.log`

5 Conclusion

The web application significantly streamlines the management of the IVOA Document Repository by automating key processes such as metadata entry, document uploads, and structured file storage. The introduction of this system eliminates the need for manual tasks like drafting and maintaining document landing pages, thereby reducing administrative overhead and minimizing errors.
Key contributions of this application include:

- **Efficient Metadata Management:** A user-friendly form interface ensures accurate and efficient metadata entry. The stored metadata is easily accessible and dynamically rendered on web pages for seamless user interaction.
- **Automated File Handling:** Uploaded files are automatically processed, renamed, and organized into a logical directory hierarchy. This eliminates inconsistencies and ensures long-term retrievability of documents.
- **Dynamic Web Pages:** By dynamically generating individual document landing pages and a centralized repository view, the application offers an up-to-date and comprehensive view of the repository at any time.

6 Future Enhancements

To further improve the application's functionality and user experience, several enhancements are planned:

- **Errata Management:** The inclusion of errata tracking will allow users to view and manage corrections or updates to published documents, ensuring the repository reflects the most accurate information.
- **Bibcode Integration:** Adding support for Bibcodes will facilitate easier integration with bibliographic databases like ADS (Astrophysics Data System), enhancing the discoverability of documents.
- **Request for Comments (RFC) Phase Integration:** By incorporating an RFC phase, the application will foster greater collaboration and transparency in the document approval process, ensuring higher-quality standards and broader acceptance within the IVOA community.

- **Advanced Search and Filtering:** Future versions will include advanced search capabilities, allowing users to filter documents by metadata attributes like author, date, or keywords.
- **User Authentication and Role Management:** Incorporating user authentication will enable role-based access control, allowing authorized users to upload or modify content while maintaining the security of the repository.

By automating previously manual workflows, this application significantly reduces administrative overhead while enhancing the accuracy and accessibility of the IVOA Document Repository. As future enhancements are integrated, the system will evolve into a fully comprehensive platform supporting IVOA's mission of interoperability and data management

A Appendix

A.1 Source Code Repository

The complete source code for this web application, including all implementation details, is available on GitHub. Repository URL: https://github.com/ivoa/ivoa-docrepo This repository contains:

- Python scripts for backend logic (Flask application).
- SQLAlchemy models for database structure.
- HTML templates and static files for frontend.
- Deployment instructions for Apache server integration.

Readers can access the code to review, test, or further enhance the functionality described in this report.

A.2 List of Dependencies:

The development and deployment of this web application required several Python libraries and other tools to implement key functionalities. Below is a categorized list of dependencies:

A.2.1 Backend Dependencies:

- **Flask:** A lightweight web framework used for routing, request handling, and rendering dynamic web pages.
- **SQLAlchemy:** An Object Relational Mapper (ORM) for managing database interactions.
- **Flask-WTF:** Provides form handling and validation capabilities.
- **SQLite3:** A lightweight, file-based database system for storing metadata.

A.2.2 Web Scraping Dependencies:

- **BeautifulSoup4:** A Python library used for parsing HTML and extracting metadata during the database population process.
- **Requests:** A library for sending HTTP requests to fetch web pages for scraping.

A.2.3 Frontend Dependencies:

- **Jinja2:** A templating engine for dynamically rendering HTML pages.
- **Bootstrap:** For responsive and consistent styling of the user interface.
- **JavaScript and jQuery:** Enhances interactivity, including AJAX requests and form validations.

A.2.4 Deployment Dependencies:

- **mod_wsgi:** A WSGI adapter module for deploying the Flask application on an Apache HTTP server.
- **Apache HTTP Server:** Used for hosting the web application.

A.2.5 Python Version:

- **Python 3.x:** The application is built using Python 3.x, ensuring compatibility with modern libraries and tools.

A.2.6 Other Tools and Utilities

- **System Libraries:** Standard libraries such as os, pathlib, tarfile, and zipfile for file management.