Google Drive Cleanup Tool

A comprehensive tool for analyzing, visualizing, and cleaning up Google Drive storage with an interactive web interface.

Project Structure

```
drive-cleanup/
├── backend/
│   ├── auth.py            # Google Drive authentication
│   ├── drive_api.py       # Core Drive API operations
│   ├── analyzer.py        # Drive structure analysis
│   ├── cleanup.py         # Deletion and cleanup operations
│   └── server.py          # API server (Flask/FastAPI)
├── frontend/
│   ├── src/
│   │   ├── App.jsx        # Main React component (visualizer)
│   │   └── index.js
│   ├── package.json
│   └── public/
├── credentials.json       # Google OAuth credentials (gitignored)
├── token.json             # User auth token (gitignored)
├── requirements.txt
└── README.md
```

Setup Instructions

1. Google Cloud Setup

1. Go to Google Cloud Console
2. Create a new project
3. Enable Google Drive API
4. Create OAuth 2.0 credentials (Desktop app)
5. Download credentials as `credentials.json`
6. Place in project root

2. Backend Setup

```
bash
```

```
# Create virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt
```

3. Frontend Setup

bash

```
cd frontend
npm install
```

Backend Code

requirements.txt

```
google-api-python-client==2.108.0
google-auth-httplib2==0.1.1
google-auth-oauthlib==1.1.0
flask==3.0.0
flask-cors==4.0.0
python-dotenv==1.0.0
```

backend/auth.py

python

```python
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from googleapiclient.discovery import build
import os
import pickle

SCOPES = ['https://www.googleapis.com/auth/drive']

def authenticate():
    """Authenticate and return Drive service"""
    creds = None

    # Token file stores user's access and refresh tokens
    if os.path.exists('token.pickle'):
        with open('token.pickle', 'rb') as token:
            creds = pickle.load(token)

    # If no valid credentials, let user log in
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file(
                'credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)

        # Save credentials for next run
        with open('token.pickle', 'wb') as token:
            pickle.dump(creds, token)

    return build('drive', 'v3', credentials=creds)
```

backend/drive_api.py

```python
```

```
from collections import defaultdict
from datetime import datetime, timedelta

def list_all_files(service):
    """Fetch all files from Drive"""
    all_files = []
    page_token = None

    while True:
        results = service.files().list(
            q="trashed=false",
            pageSize=1000,
            fields="nextPageToken, files(id, name, mimeType, parents, size, createdTime, modifiedTime, webViewLink)",
            pageToken=page_token
        ).execute()

        all_files.extend(results.get('files', []))
        page_token = results.get('nextPageToken')

        if not page_token:
            break

    return all_files

def build_tree_structure(all_files):
    """Build parent-child relationships and calculate folder sizes"""

    file_map = {f['id']: f for f in all_files}
    children_map = defaultdict(list)

    # Build parent-child relationships
    for file in all_files:
        parents = file.get('parents', [])
        for parent in parents:
            children_map[parent].append(file['id'])

    # Calculate folder sizes recursively
    def calc_size(file_id):
        file = file_map.get(file_id)
        if not file:
            return 0

        # Files have direct size
```

```
        if file['mimeType'] != 'application/vnd.google-apps.folder':
            return int(file.get('size', 0))

        # Folders need to sum children
        total = 0
        for child_id in children_map.get(file_id, []):
            total += calc_size(child_id)

        file['calculatedSize'] = total
        return total

    # Calculate sizes for all root items
    roots = [f for f in all_files if not f.get('parents')]
    for root in roots:
        calc_size(root['id'])

    return {
        'files': all_files,
        'file_map': file_map,
        'children_map': dict(children_map)
    }

def search_files(service, query):
    """Search files with custom query"""
    results = service.files().list(
        q=query,
        pageSize=100,
        fields="files(id, name, mimeType, size, modifiedTime, webViewLink)"
    ).execute()

    return results.get('files', [])

def get_file_metadata(service, file_id):
    """Get detailed metadata for a specific file"""
    return service.files().get(
        fileId=file_id,
        fields="*"
    ).execute()

def create_folder(service, name, parent_id=None):
    """Create a new folder"""
    file_metadata = {
        'name': name,
        'mimeType': 'application/vnd.google-apps.folder'
```

```
    }
    if parent_id:
        file_metadata['parents'] = [parent_id]

    folder = service.files().create(
        body=file_metadata,
        fields='id'
    ).execute()

    return folder.get('id')

def move_file(service, file_id, new_parent_id):
    """Move file to different folder"""
    file = service.files().get(
        fileId=file_id,
        fields='parents'
    ).execute()

    previous_parents = ",".join(file.get('parents', []))

    service.files().update(
        fileId=file_id,
        addParents=new_parent_id,
        removeParents=previous_parents,
        fields='id, parents'
    ).execute()
```

backend/analyzer.py

python

```
from datetime import datetime, timedelta
from collections import defaultdict

def analyze_drive(all_files):
    """Analyze Drive and identify problem areas"""

    problems = {
        'large_videos': [],
        'old_large_files': [],
        'duplicates': defaultdict(list),
        'deep_nesting': [],
        'empty_folders': []
    }

    two_years_ago = datetime.now() - timedelta(days=730)

    for file in all_files:
        size = int(file.get('size', 0))
        mime = file['mimeType']
        name = file['name']

        # Large videos (>500MB)
        if 'video' in mime and size > 500_000_000:
            problems['large_videos'].append({
                'id': file['id'],
                'name': name,
                'size': size,
                'modified': file.get('modifiedTime'),
                'link': file.get('webViewLink')
            })

        # Old large files (>100MB, >2 years)
        if size > 100_000_000 and file.get('modifiedTime'):
            mod_time = datetime.fromisoformat(
                file['modifiedTime'].replace('Z', '+00:00')
            )
            if mod_time < two_years_ago:
                problems['old_large_files'].append({
                    'id': file['id'],
                    'name': name,
                    'size': size,
                    'modified': file.get('modifiedTime'),
                    'link': file.get('webViewLink')
```

```
            })

        # Track duplicates by name
        problems['duplicates'][name].append(file)

    # Filter to actual duplicates
    problems['duplicates'] = {
        name: files
        for name, files in problems['duplicates'].items()
        if len(files) > 1
    }

    # Calculate statistics
    stats = {
        'total_files': len(all_files),
        'total_size': sum(int(f.get('size', 0)) for f in all_files),
        'video_count': len([f for f in all_files if 'video' in f['mimeType']]),
        'video_size': sum(
            int(f.get('size', 0))
            for f in all_files
            if 'video' in f['mimeType']
        ),
        'folder_count': len([
            f for f in all_files
            if f['mimeType'] == 'application/vnd.google-apps.folder'
        ])
    }

    return {
        'problems': problems,
        'stats': stats
    }

def find_empty_folders(service, all_files):
    """Find folders with no children"""
    folders = [
        f for f in all_files
        if f['mimeType'] == 'application/vnd.google-apps.folder'
    ]

    empty = []
    for folder in folders:
        children = service.files().list(
            q=f"'{folder['id']}' in parents and trashed=false",
```

```
            pageSize=1
        ).execute()

        if not children.get('files'):
            empty.append(folder)

    return empty

def calculate_depth(file_id, file_map, children_map, depth=0):
    """Calculate nesting depth of folders"""
    if depth > 10:  # Prevent infinite recursion
        return depth

    file = file_map.get(file_id)
    if not file:
        return depth

    parents = file.get('parents', [])
    if not parents:
        return depth

    return max(
        calculate_depth(parent, file_map, children_map, depth + 1)
        for parent in parents
    )
```

backend/cleanup.py

python

```python
def delete_file(service, file_id):
    """Permanently delete a file"""
    try:
        service.files().delete(fileId=file_id).execute()
        return {'success': True, 'file_id': file_id}
    except Exception as e:
        return {'success': False, 'file_id': file_id, 'error': str(e)}

def trash_file(service, file_id):
    """Move file to trash (recoverable)"""
    try:
        service.files().update(
            fileId=file_id,
            body={'trashed': True}
        ).execute()
        return {'success': True, 'file_id': file_id}
    except Exception as e:
        return {'success': False, 'file_id': file_id, 'error': str(e)}

def batch_delete(service, file_ids):
    """Delete multiple files in batch"""
    from googleapiclient.http import BatchHttpRequest

    results = []

    def callback(request_id, response, exception):
        if exception:
            results.append({
                'success': False,
                'file_id': request_id,
                'error': str(exception)
            })
        else:
            results.append({
                'success': True,
                'file_id': request_id
            })

    batch = service.new_batch_http_request(callback=callback)

    for file_id in file_ids:
        batch.add(service.files().delete(fileId=file_id), request_id=file_id)
```

```
    batch.execute()
    return results

def batch_trash(service, file_ids):
    """Move multiple files to trash in batch"""
    results = []

    for file_id in file_ids:
        result = trash_file(service, file_id)
        results.append(result)

    return results

def empty_trash(service):
    """Permanently delete all trashed files"""
    try:
        service.files().emptyTrash().execute()
        return {'success': True}
    except Exception as e:
        return {'success': False, 'error': str(e)}

def delete_duplicates(service, duplicates, keep='newest'):
    """Delete duplicate files, keeping one copy"""
    to_delete = []

    for name, files in duplicates.items():
        # Sort by modified time
        files_sorted = sorted(
            files,
            key=lambda x: x.get('modifiedTime', ''),
            reverse=(keep == 'newest')
        )

        # Keep first, delete rest
        to_delete.extend([f['id'] for f in files_sorted[1:]])

    return batch_trash(service, to_delete)

def delete_old_large_files(service, min_size_bytes, days_old):
    """Delete files larger than threshold and older than days"""
    from datetime import datetime, timedelta

    cutoff = datetime.now() - timedelta(days=days_old)
    cutoff_str = cutoff.isoformat() + 'Z'
```

```
    query = f"modifiedTime < '{cutoff_str}' and trashed=false"

    results = service.files().list(
        q=query,
        pageSize=1000,
        fields="files(id, name, size, modifiedTime)"
    ).execute()

    files = results.get('files', [])
    large_old = [
        f for f in files
        if int(f.get('size', 0)) > min_size_bytes
    ]

    file_ids = [f['id'] for f in large_old]
    return batch_trash(service, file_ids)
```

backend/server.py

python

```python
from flask import Flask, jsonify, request
from flask_cors import CORS
from auth import authenticate
from drive_api import list_all_files, build_tree_structure
from analyzer import analyze_drive, find_empty_folders
from cleanup import (
    delete_file, trash_file, batch_delete, batch_trash,
    empty_trash, delete_duplicates, delete_old_large_files
)

app = Flask(__name__)
CORS(app)

# Initialize Drive service
service = authenticate()

@app.route('/api/scan', methods=['GET'])
def scan_drive():
    """Scan entire Drive and return structure"""
    files = list_all_files(service)
    tree = build_tree_structure(files)
    analysis = analyze_drive(files)

    return jsonify({
        'files': tree['files'],
        'children_map': tree['children_map'],
        'analysis': analysis
    })

@app.route('/api/delete', methods=['POST'])
def delete_files():
    """Delete specific files"""
    data = request.json
    file_ids = data.get('file_ids', [])
    permanent = data.get('permanent', False)

    if permanent:
        results = batch_delete(service, file_ids)
    else:
        results = batch_trash(service, file_ids)

    return jsonify({'results': results})
```

```python
@app.route('/api/cleanup/videos', methods=['POST'])
def cleanup_videos():
    """Delete old large videos"""
    data = request.json
    min_size_gb = data.get('min_size_gb', 1)
    days_old = data.get('days_old', 365)

    min_bytes = min_size_gb * 1024 * 1024 * 1024
    results = delete_old_large_files(service, min_bytes, days_old)

    return jsonify({'results': results})

@app.route('/api/cleanup/duplicates', methods=['POST'])
def cleanup_duplicates():
    """Delete duplicate files"""
    data = request.json
    duplicates = data.get('duplicates', {})
    keep = data.get('keep', 'newest')

    results = delete_duplicates(service, duplicates, keep)

    return jsonify({'results': results})

@app.route('/api/empty-trash', methods=['POST'])
def api_empty_trash():
    """Permanently empty trash"""
    result = empty_trash(service)
    return jsonify(result)

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

Frontend Code

frontend/package.json

```json
```

```
{
  "name": "drive-cleanup-frontend",
  "version": "1.0.0",
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "lucide-react": "^0.263.1",
    "axios": "^1.6.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build"
  },
  "devDependencies": {
    "react-scripts": "5.0.1"
  }
}
```

frontend/src/App.jsx

(Use the React component from the artifact above - it's the complete DriveVisualizer component)

Usage

Start Backend

```
bash

python backend/server.py
```

Start Frontend

```
bash

cd frontend
npm start
```

API Endpoints

- `GET /api/scan` - Scan and analyze Drive
- `POST /api/delete` - Delete files by ID
- `POST /api/cleanup/videos` - Clean up old videos
- `POST /api/cleanup/duplicates` - Remove duplicates

- `POST /api/empty-trash` - Empty trash permanently

Features

Current

- ✅ Complete Drive structure visualization
- ✅ Treemap and list views
- ✅ Find large videos
- ✅ Find duplicates
- ✅ Find old files
- ✅ Batch deletion
- ✅ Trash management

Roadmap

- [ ] Real-time sync
- [ ] Custom tagging system
- [ ] Smart auto-organization
- [ ] Storage timeline view
- [ ] Export reports
- [ ] Undo functionality
- [ ] File preview

Safety Notes

1. **Always test with trash first** before permanent deletion
2. **Backup important files** before running cleanup operations
3. **Review lists carefully** before confirming deletions
4. **Check API quotas** - Drive API has daily limits
5. **Keep credentials secure** - Never commit `credentials.json` or `token.json`

Git Configuration

`.gitignore`

```
# Credentials
credentials.json
```

```
token.json
token.pickle

# Python
__pycache__/
*.py[cod]
*$py.class
venv/
*.so
.Python

# Node
node_modules/
npm-debug.log*
build/

# IDE
.vscode/
.idea/
*.swp
*.swo

# OS
.DS_Store
Thumbs.db
```

License

MIT

Contributing

Pull requests welcome! Please ensure:

- Code follows PEP 8 (Python) and Airbnb style (JavaScript)
- All API calls include error handling
- User confirmation required for destructive operations