import pandas as pd

import numpy as np

import datacompy

import xlsxwriter

from datetime import datetime

import os

import zipfile

from io import BytesIO

import logging

from typing import Dict, List, Optional, Tuple

from backend.profile\_utils import (

    generate\_distribution\_plot,

    generate\_frequency\_plot,

    generate\_comparison\_plot,

    generate\_comparison\_rows,

    calculate\_column\_stats

)

logger = logging.getLogger(\_\_name\_\_)

def generate\_mismatch\_html(comparison):

    """Generate HTML for mismatches with proper error handling."""

    try:

        mismatches\_html = []

        # Handle source-only rows

        if hasattr(comparison, 'df1\_unq\_rows') and not comparison.df1\_unq\_rows.empty:

            mismatches\_html.append("<h3>Rows Only in Source</h3>")

            mismatches\_html.append(comparison.df1\_unq\_rows.head(5).to\_html())

        # Handle target-only rows

        if hasattr(comparison, 'df2\_unq\_rows') and not comparison.df2\_unq\_rows.empty:

            mismatches\_html.append("<h3>Rows Only in Target</h3>")

            mismatches\_html.append(comparison.df2\_unq\_rows.head(5).to\_html())

        if mismatches\_html:

            return "\n".join(mismatches\_html)

        return "<p>No mismatches found</p>"

    except Exception as e:

        logger.warning(f"Error generating mismatch HTML: {str(e)}")

        return f"<p>Error generating mismatch details: {str(e)}</p>"

def generate\_datacompy\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                            join\_columns: List[str], mapping\_df: pd.DataFrame,

                            join\_mappings: Dict[str, str]) -> Tuple[BytesIO, BytesIO]:

    """Generate a DataCompy comparison report."""

    try:

        # Get excluded columns

        excluded\_columns = mapping\_df[mapping\_df['Exclude from Comparison']]['Source Column'].tolist()

        # Create mapping dictionary

        column\_mapping = dict(zip(

            mapping\_df['Source Column'],

            mapping\_df['Target Column']

        ))

        # Filter out unmapped and excluded columns

        valid\_columns = {

            src: tgt for src, tgt in column\_mapping.items()

            if tgt and not pd.isna(tgt) and src not in excluded\_columns

        }

        # Get all columns that need to be processed

        source\_cols = list(valid\_columns.keys())

        target\_cols = [valid\_columns[src] for src in source\_cols]

        # Create unique column names for comparison

        source\_rename = {}

        target\_rename = {}

        final\_join\_columns = []

        # First, handle join columns to ensure they're unique

        for src\_col in join\_columns:

            tgt\_col = join\_mappings[src\_col]

            unique\_join\_name = f"join\_{src\_col}"

            source\_rename[src\_col] = unique\_join\_name

            target\_rename[tgt\_col] = unique\_join\_name

            final\_join\_columns.append(unique\_join\_name)

        # Then handle comparison columns

        for src\_col, tgt\_col in valid\_columns.items():

            if src\_col not in join\_columns:  # Skip if it's already a join column

                unique\_col\_name = f"compare\_{src\_col}"

                source\_rename[src\_col] = unique\_col\_name

                target\_rename[tgt\_col] = unique\_col\_name

        # Create copies of DataFrames with only needed columns

        source\_compare = source\_df[list(set(source\_cols + join\_columns))].copy()

        target\_compare = target\_df[list(set(target\_cols + [join\_mappings[src] for src in join\_columns]))].copy()

        # Rename columns to ensure uniqueness

        source\_compare.rename(columns=source\_rename, inplace=True)

        target\_compare.rename(columns=target\_rename, inplace=True)

        # Update join columns for the comparison

        join\_columns = final\_join\_columns

        logger.info(f"Source columns after renaming: {source\_compare.columns.tolist()}")

        logger.info(f"Target columns after renaming: {target\_compare.columns.tolist()}")

        logger.info(f"Join columns for comparison: {join\_columns}")

        # Create comparison object with proper configuration

        try:

            comparison = datacompy.Compare(

                df1=source\_compare,

                df2=target\_compare,

                join\_columns=join\_columns,

                df1\_name='Source',

                df2\_name='Target',

                on\_index=False

            )

        except Exception as e:

            logger.warning(f"Initial comparison failed: {str(e)}, trying with string conversion")

            # Convert all columns to string type for comparison

            source\_compare = source\_compare.astype(str)

            target\_compare = target\_compare.astype(str)

            comparison = datacompy.Compare(

                df1=source\_compare,

                df2=target\_compare,

                join\_columns=join\_columns,

                df1\_name='Source',

                df2\_name='Target',

                on\_index=False

            )

        # Generate Excel report

        excel\_output = BytesIO()

        with pd.ExcelWriter(excel\_output, engine='xlsxwriter') as writer:

            # Write summary

            summary\_data = {

                'Metric': [

                    'Rows in Source',

                    'Rows in Target',

                    'Rows in Common',

                    'Rows Only in Source',

                    'Rows Only in Target',

                    'Columns Match',

                    'All Row Values Match'

                ],

                'Value': [

                    len(source\_compare),

                    len(target\_compare),

                    comparison.intersect\_rows,

                    len(comparison.df1\_unq\_rows) if hasattr(comparison, 'df1\_unq\_rows') else 0,

                    len(comparison.df2\_unq\_rows) if hasattr(comparison, 'df2\_unq\_rows') else 0,

                    comparison.all\_columns\_match(),

                    comparison.matches()

                ]

            }

            pd.DataFrame(summary\_data).to\_excel(writer, sheet\_name='Summary', index=False)

            # Write column stats

            if hasattr(comparison, 'column\_stats'):

                comparison.column\_stats.to\_excel(writer, sheet\_name='Column Stats', index=True)

            # Write sample mismatches with proper error handling

            try:

                mismatches\_df = pd.DataFrame()

                # Get source-only rows

                if hasattr(comparison, 'df1\_unq\_rows') and not comparison.df1\_unq\_rows.empty:

                    source\_mismatches = comparison.df1\_unq\_rows.head(5).copy()

                    source\_mismatches['Match Type'] = 'Source Only'

                    mismatches\_df = pd.concat([mismatches\_df, source\_mismatches])

                # Get target-only rows

                if hasattr(comparison, 'df2\_unq\_rows') and not comparison.df2\_unq\_rows.empty:

                    target\_mismatches = comparison.df2\_unq\_rows.head(5).copy()

                    target\_mismatches['Match Type'] = 'Target Only'

                    mismatches\_df = pd.concat([mismatches\_df, target\_mismatches])

                if not mismatches\_df.empty:

                    mismatches\_df.to\_excel(writer, sheet\_name='Sample Mismatches', index=True)

                else:

                    pd.DataFrame({'Status': ['No mismatches found']}).to\_excel(

                        writer, sheet\_name='Sample Mismatches', index=False)

            except Exception as e:

                logger.warning(f"Error generating sample mismatches: {str(e)}")

                pd.DataFrame({'Error': [f'Failed to generate sample mismatches: {str(e)}']}).to\_excel(

                    writer, sheet\_name='Sample Mismatches', index=False)

        # Generate HTML report

        html\_output = BytesIO()

        html\_report = f"""

        <html>

        <head>

            <title>DataCompy Comparison Report</title>

            <style>

                body {{ font-family: Arial, sans-serif; margin: 20px; }}

                table {{ border-collapse: collapse; width: 100%; margin-bottom: 20px; }}

                th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}

                th {{ background-color: #f2f2f2; }}

                .pass {{ color: green; }}

                .fail {{ color: red; }}

                .section {{ margin-bottom: 30px; }}

            </style>

        </head>

        <body>

            <h1>DataCompy Comparison Report</h1>

            <div class="section">

                <h2>Summary</h2>

                <p>Source rows: {len(source\_compare)}</p>

                <p>Target rows: {len(target\_compare)}</p>

                <p>Rows in common: {comparison.intersect\_rows}</p>

                <p>Rows only in source: {len(comparison.df1\_unq\_rows) if hasattr(comparison, 'df1\_unq\_rows') else 0}</p>

                <p>Rows only in target: {len(comparison.df2\_unq\_rows) if hasattr(comparison, 'df2\_unq\_rows') else 0}</p>

                <p>Columns match: <span class="{'pass' if comparison.all\_columns\_match() else 'fail'}">{comparison.all\_columns\_match()}</span></p>

                <p>All rows match: <span class="{'pass' if comparison.matches() else 'fail'}">{comparison.matches()}</span></p>

            </div>

            <div class="section">

                <h2>Column Statistics</h2>

                {comparison.column\_stats.to\_html() if hasattr(comparison, 'column\_stats') else '<p>No column statistics available</p>'}

            </div>

            <div class="section">

                <h2>Sample Mismatches</h2>

                {generate\_mismatch\_html(comparison)}

            </div>

        </body>

        </html>

        """

        html\_output.write(html\_report.encode('utf-8'))

        html\_output.seek(0)

        excel\_output.seek(0)

        return excel\_output, html\_output

    except Exception as e:

        logger.error(f"Error generating DataCompy report: {str(e)}")

        raise Exception(f"Failed to generate DataCompy report: {str(e)}")

def generate\_ydata\_profile(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                         mapping\_df: pd.DataFrame) -> Tuple[BytesIO, BytesIO, BytesIO]:

    """Generate column-wise Y-Data Profiling reports."""

    try:

        # Create mapping dictionary

        column\_mapping = dict(zip(

            mapping\_df['Source Column'],

            mapping\_df['Target Column']

        ))

        # Filter out unmapped and excluded columns

        excluded\_columns = mapping\_df[mapping\_df['Exclude from Comparison']]['Source Column'].tolist()

        valid\_columns = {

            src: tgt for src, tgt in column\_mapping.items()

            if tgt and not pd.isna(tgt) and src not in excluded\_columns

        }

        # Initialize output buffers

        source\_output = BytesIO()

        target\_output = BytesIO()

        comparison\_output = BytesIO()

        # Start HTML reports with headers and CSS

        html\_header = """

            <html><head><title>{title}</title>

            <style>

                body { font-family: Arial, sans-serif; margin: 20px; }

                .column-profile { margin-bottom: 40px; padding: 20px; border: 1px solid #ddd; border-radius: 5px; }

                .stats-table { width: 100%; border-collapse: collapse; margin-top: 10px; }

                .stats-table th, .stats-table td { border: 1px solid #ddd; padding: 8px; text-align: left; }

                .stats-table th { background-color: #f5f5f5; }

                .visualization { margin-top: 20px; }

                .chart { width: 100%; max-width: 800px; margin: 20px 0; }

                .header { background-color: #f8f9fa; padding: 20px; margin-bottom: 20px; border-radius: 5px; }

                .summary { margin: 20px 0; }

                .metric { display: inline-block; margin: 10px; padding: 15px; background: #fff; border: 1px solid #ddd; border-radius: 5px; }

                .diff { background-color: #ffe6e6; }

                .match { background-color: #e6ffe6; }

            </style>

            </head><body>

            <div class="header">

                <h1>{title}</h1>

                <p>Generated on: {timestamp}</p>

            </div>

        """

        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        source\_html = [html\_header.format(title="Source Data Profile", timestamp=timestamp)]

        target\_html = [html\_header.format(title="Target Data Profile", timestamp=timestamp)]

        comparison\_html = [html\_header.format(title="Column Comparison Profile", timestamp=timestamp)]

        # Process each column pair

        for src\_col, tgt\_col in valid\_columns.items():

            logger.info(f"Processing column pair: {src\_col} -> {tgt\_col}")

            # Get column data

            source\_data = source\_df[src\_col]

            target\_data = target\_df[tgt\_col]

            # Calculate statistics

            source\_stats = calculate\_column\_stats(source\_data)

            target\_stats = calculate\_column\_stats(target\_data)

            # Generate HTML for source column

            source\_html.append(f"""

                <div class="column-profile">

                    <h2>Column: {src\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Type:</strong> {source\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Unique Values:</strong> {source\_stats['Unique Values']}

                        </div>

                        <div class="metric">

                            <strong>Missing %:</strong> {source\_stats['Missing %']:.2f}%

                        </div>

                    </div>

                    <table class="stats-table">

                        <tr><th>Metric</th><th>Value</th></tr>

                        {

                            ''.join([f"<tr><td>{k}</td><td>{v if isinstance(v, str) else f'{v:.2f}' if isinstance(v, float) else v}</td></tr>"

                                    for k, v in source\_stats.items()])

                        }

                    </table>

                    <div class="visualization">

                        <h3>Value Distribution</h3>

                        {generate\_distribution\_plot(source\_data) if pd.api.types.is\_numeric\_dtype(source\_data) else generate\_frequency\_plot(source\_data)}

                    </div>

                </div>

            """)

            # Generate HTML for target column

            target\_html.append(f"""

                <div class="column-profile">

                    <h2>Column: {tgt\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Type:</strong> {target\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Unique Values:</strong> {target\_stats['Unique Values']}

                        </div>

                        <div class="metric">

                            <strong>Missing %:</strong> {target\_stats['Missing %']:.2f}%

                        </div>

                    </div>

                    <table class="stats-table">

                        <tr><th>Metric</th><th>Value</th></tr>

                        {

                            ''.join([f"<tr><td>{k}</td><td>{v if isinstance(v, str) else f'{v:.2f}' if isinstance(v, float) else v}</td></tr>"

                                    for k, v in target\_stats.items()])

                        }

                    </table>

                    <div class="visualization">

                        <h3>Value Distribution</h3>

                        {generate\_distribution\_plot(target\_data) if pd.api.types.is\_numeric\_dtype(target\_data) else generate\_frequency\_plot(target\_data)}

                    </div>

                </div>

            """)

            # Generate comparison HTML

            comparison\_html.append(f"""

                <div class="comparison-profile">

                    <h2>Column Comparison: {src\_col} ↔ {tgt\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Source Type:</strong> {source\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Target Type:</strong> {target\_data.dtype}

                        </div>

                    </div>

                    <table class="comparison-table">

                        <tr>

                            <th>Metric</th>

                            <th>Source</th>

                            <th>Target</th>

                            <th>Difference</th>

                        </tr>

                        {generate\_comparison\_rows(source\_stats, target\_stats)}

                    </table>

                    <div class="visualization">

                        <h3>Distribution Comparison</h3>

                        {generate\_comparison\_plot(source\_data, target\_data)}

                    </div>

                </div>

            """)

        # Close HTML documents

        source\_html.append("</body></html>")

        target\_html.append("</body></html>")

        comparison\_html.append("</body></html>")

        # Write to output buffers

        source\_output.write(''.join(source\_html).encode('utf-8'))

        target\_output.write(''.join(target\_html).encode('utf-8'))

        comparison\_output.write(''.join(comparison\_html).encode('utf-8'))

        # Reset buffer positions

        source\_output.seek(0)

        target\_output.seek(0)

        comparison\_output.seek(0)

        return source\_output, target\_output, comparison\_output

    except Exception as e:

        logger.error(f"Error generating Y-Data profile: {str(e)}")

        raise Exception(f"Failed to generate Y-Data profile: {str(e)}")

def create\_consolidated\_report(datacompy\_excel: BytesIO,

                             datacompy\_html: BytesIO,

                             ydata\_source\_report: BytesIO,

                             ydata\_target\_report: BytesIO,

                             ydata\_comparison\_report: BytesIO,

                             regression\_report: BytesIO,

                             difference\_report: BytesIO) -> BytesIO:

    """Combine all reports into a single consolidated report."""

    try:

        timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')

        output = BytesIO()

        with zipfile.ZipFile(output, 'w', zipfile.ZIP\_DEFLATED) as zf:

            # Add all reports to a single consolidated file

            zf.writestr(f'reports/datacompy\_report\_{timestamp}.xlsx',

                       datacompy\_excel.getvalue())

            zf.writestr(f'reports/datacompy\_report\_{timestamp}.html',

                       datacompy\_html.getvalue())

            zf.writestr(f'reports/ydata\_source\_profile\_{timestamp}.html',

                       ydata\_source\_report.getvalue())

            zf.writestr(f'reports/ydata\_target\_profile\_{timestamp}.html',

                       ydata\_target\_report.getvalue())

            zf.writestr(f'reports/ydata\_comparison\_profile\_{timestamp}.html',

                       ydata\_comparison\_report.getvalue())

            zf.writestr(f'reports/regression\_report\_{timestamp}.xlsx',

                       regression\_report.getvalue())

            zf.writestr(f'reports/difference\_report\_{timestamp}.xlsx',

                       difference\_report.getvalue())

            # Add a summary README

            readme\_content = f"""

            Consolidated Data Comparison Report

            This ZIP file contains the following reports:

            1. datacompy\_report - Detailed comparison of datasets

            2. ydata\_profile - Column-wise data profiling

            3. regression\_report - Aggregation, count, and distinct value checks

            4. difference\_report - Side-by-side differences

            Generated on: {timestamp}

            """

            zf.writestr('README.txt', readme\_content.strip())

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error creating consolidated report: {str(e)}")

        raise Exception(f"Failed to create consolidated report: {str(e)}")

def generate\_regression\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                            mapping\_df: pd.DataFrame, dtype\_mapping: Dict[str, str]) -> BytesIO:

    """Generate Excel-based regression report with multiple tabs."""

    try:

        output = BytesIO()

        with pd.ExcelWriter(output, engine='xlsxwriter') as writer:

            workbook = writer.book

            # Create formats for PASS/FAIL cells

            pass\_format = workbook.add\_format({'bg\_color': '#90EE90'})  # Light green

            fail\_format = workbook.add\_format({'bg\_color': '#FFB6C6'})  # Light pink

            # Generate AggregationCheck tab

            \_generate\_aggregation\_check(source\_df, target\_df, mapping\_df, writer,

                                     pass\_format, fail\_format)

            # Generate CountCheck tab

            \_generate\_count\_check(source\_df, target\_df, writer, pass\_format, fail\_format)

            # Generate DistinctCheck tab

            \_generate\_distinct\_check(source\_df, target\_df, mapping\_df, writer,

                                  pass\_format, fail\_format)

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error generating regression report: {str(e)}")

        raise Exception(f"Failed to generate regression report: {str(e)}")

def \_generate\_aggregation\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                              mapping\_df: pd.DataFrame, writer: pd.ExcelWriter,

                              pass\_format: xlsxwriter.format.Format,

                              fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the AggregationCheck tab in the regression report."""

    # Get numeric columns

    numeric\_cols = source\_df.select\_dtypes(include=[np.number]).columns

    results = []

    for col in numeric\_cols:

        if col in mapping\_df['Source Column'].values:

            target\_col = mapping\_df[mapping\_df['Source Column'] == col]['Target Column'].iloc[0]

            source\_sum = source\_df[col].sum()

            target\_sum = target\_df[target\_col].sum()

            match = np.isclose(source\_sum, target\_sum, rtol=1e-05)

            results.append({

                'Source Column': col,

                'Target Column': target\_col,

                'Source Sum': source\_sum,

                'Target Sum': target\_sum,

                'Result': 'PASS' if match else 'FAIL'

            })

    # Create DataFrame and write to Excel

    agg\_df = pd.DataFrame(results)

    if not agg\_df.empty:

        agg\_df.to\_excel(writer, sheet\_name='AggregationCheck', index=False)

        # Apply conditional formatting

        worksheet = writer.sheets['AggregationCheck']

        result\_col = agg\_df.columns.get\_loc('Result')

        for row in range(len(agg\_df)):

            if agg\_df.iloc[row]['Result'] == 'PASS':

                worksheet.write(row + 1, result\_col, 'PASS', pass\_format)

            else:

                worksheet.write(row + 1, result\_col, 'FAIL', fail\_format)

    else:

        pd.DataFrame({'Message': ['No numeric columns to compare']}).to\_excel(

            writer, sheet\_name='AggregationCheck', index=False)

def \_generate\_count\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                         writer: pd.ExcelWriter,

                         pass\_format: xlsxwriter.format.Format,

                         fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the CountCheck tab in the regression report."""

    count\_data = {

        'Source File Name': source\_df.name if hasattr(source\_df, 'name') else 'Source',

        'Target File Name': target\_df.name if hasattr(target\_df, 'name') else 'Target',

        'Source Count': len(source\_df),

        'Target Count': len(target\_df),

        'Result': 'PASS' if len(source\_df) == len(target\_df) else 'FAIL'

    }

    count\_df = pd.DataFrame([count\_data])

    count\_df.to\_excel(writer, sheet\_name='CountCheck', index=False)

    # Apply conditional formatting

    worksheet = writer.sheets['CountCheck']

    result\_col = count\_df.columns.get\_loc('Result')

    if count\_data['Result'] == 'PASS':

        worksheet.write(1, result\_col, 'PASS', pass\_format)

    else:

        worksheet.write(1, result\_col, 'FAIL', fail\_format)

def \_generate\_distinct\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                           mapping\_df: pd.DataFrame, writer: pd.ExcelWriter,

                           pass\_format: xlsxwriter.format.Format,

                           fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the DistinctCheck tab in the regression report."""

    # Get non-numeric columns

    non\_numeric\_cols = source\_df.select\_dtypes(exclude=[np.number]).columns

    results = []

    for col in non\_numeric\_cols:

        if col in mapping\_df['Source Column'].values:

            target\_col = mapping\_df[mapping\_df['Source Column'] == col]['Target Column'].iloc[0]

            source\_distinct = set(source\_df[col].dropna().unique())

            target\_distinct = set(target\_df[target\_col].dropna().unique())

            source\_count = len(source\_distinct)

            target\_count = len(target\_distinct)

            count\_match = source\_count == target\_count

            values\_match = source\_distinct == target\_distinct

            results.append({

                'Source Column': col,

                'Target Column': target\_col,

                'Source Distinct Count': source\_count,

                'Target Distinct Count': target\_count,

                'Count Match': 'PASS' if count\_match else 'FAIL',

                'Values Match': 'PASS' if values\_match else 'FAIL',

                'Source Distinct Values': ', '.join(map(str, sorted(source\_distinct))),

                'Target Distinct Values': ', '.join(map(str, sorted(target\_distinct)))

            })

    # Create DataFrame and write to Excel

    distinct\_df = pd.DataFrame(results)

    if not distinct\_df.empty:

        distinct\_df.to\_excel(writer, sheet\_name='DistinctCheck', index=False)

        # Apply conditional formatting

        worksheet = writer.sheets['DistinctCheck']

        count\_match\_col = distinct\_df.columns.get\_loc('Count Match')

        values\_match\_col = distinct\_df.columns.get\_loc('Values Match')

        for row in range(len(distinct\_df)):

            if distinct\_df.iloc[row]['Count Match'] == 'PASS':

                worksheet.write(row + 1, count\_match\_col, 'PASS', pass\_format)

            else:

                worksheet.write(row + 1, count\_match\_col, 'FAIL', fail\_format)

            if distinct\_df.iloc[row]['Values Match'] == 'PASS':

                worksheet.write(row + 1, values\_match\_col, 'PASS', pass\_format)

            else:

                worksheet.write(row + 1, values\_match\_col, 'FAIL', fail\_format)

    else:

        pd.DataFrame({'Message': ['No non-numeric columns to compare']}).to\_excel(

            writer, sheet\_name='DistinctCheck', index=False)

def create\_individual\_reports\_zip(datacompy\_excel: BytesIO,

                                datacompy\_html: BytesIO,

                                ydata\_source\_report: BytesIO,

                                ydata\_target\_report: BytesIO,

                                ydata\_comparison\_report: BytesIO,

                                regression\_report: BytesIO,

                                difference\_report: BytesIO) -> BytesIO:

    """Create a ZIP file containing individual reports in separate folders."""

    try:

        timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')

        output = BytesIO()

        with zipfile.ZipFile(output, 'w', zipfile.ZIP\_DEFLATED) as zf:

            # DataCompy Reports

            zf.writestr(f'datacompy/datacompy\_report\_{timestamp}.xlsx',

                       datacompy\_excel.getvalue())

            zf.writestr(f'datacompy/datacompy\_report\_{timestamp}.html',

                       datacompy\_html.getvalue())

            # Y-Data Profiles

            zf.writestr(f'ydata\_profile/source\_profile\_{timestamp}.html',

                       ydata\_source\_report.getvalue())

            zf.writestr(f'ydata\_profile/target\_profile\_{timestamp}.html',

                       ydata\_target\_report.getvalue())

            zf.writestr(f'ydata\_profile/comparison\_profile\_{timestamp}.html',

                       ydata\_comparison\_report.getvalue())

            # Regression Report

            zf.writestr(f'regression/regression\_report\_{timestamp}.xlsx',

                       regression\_report.getvalue())

            # Difference Report

            zf.writestr(f'differences/difference\_report\_{timestamp}.xlsx',

                       difference\_report.getvalue())

            # Add a README file

            readme\_content = f"""

            Data Comparison Reports

            1. datacompy/ - Contains detailed comparison report

            2. ydata\_profile/ - Contains column-wise data profiling

            3. regression/ - Contains aggregation, count, and distinct value checks

            4. differences/ - Contains side-by-side difference report

            Generated on: {timestamp}

            """

            zf.writestr('README.txt', readme\_content.strip())

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error creating individual reports zip: {str(e)}")

        raise Exception(f"Failed to create individual reports zip: {str(e)}")