Python App

Carmelo Costanzo

March 2025

1 Skeleton

```
dash_app
 __init__.py              # (Empty or with a short package docstring)
 main.py                  # Entry point (imports layout and callbacks)
 config_and_data.py       # Contains configuration loading, markdown, Excel data, etc.
 utils.py                 # Contains configuration loading, markdown, Excel data, etc.
 layout.py                # Contains only the layout code (see below)
 callbacks                # Folder with all callback modules
    __init__.py          # Aggregates and registers all callbacks
    tables.py            # Contains all your callbacks (or split into multiple modules as ne
    plots.py             # Contains all your callbacks (or split into multiple modules as ne
    modals.py            # Contains all your callbacks (or split into multiple modules as ne
    ppt.py               # Contains all your callbacks (or split into multiple modules as ne
 assets                   # Contains static assets (CSS, images, Markdown files, etc.)
     Amundi.jpg
     Institute_View.md
     FRV.md
     MacroEdge_Dashboard.xlsx
     bbg_per_country.xlsx
     MacroEdge.xlsx
config.json                 # Your JSON configuration file
```

2 init.py

```
"""
MacroEdge Dashboard Application

A Dash-based dashboard for visualizing financial and economic data,
including fundamentals, risks, and valuations for various countries.
"""

__version__ = '1.0.0'
```

3 main.py

```
"""
Application entry point - initializes the Dash app, sets up caching,
registers callbacks, and runs the server.
"""

import os
import subprocess
from datetime import datetime
from flask_caching import Cache

# Import the app from layout module
from .layout import app

# Local imports
from .config_and_data import config, get_dashboard_file_path

# Register all callbacks (this imports all callback modules)
from .callbacks import register_callbacks

# Setup cache based on configuration
cache_config = config["app_settings"]["cache"]
cache = Cache(app.server, config={
    'CACHE_TYPE': cache_config["type"],
    'CACHE_DIR': cache_config["directory"],
    'CACHE_DEFAULT_TIMEOUT': cache_config["default_timeout"]
})

# Register all callbacks
register_callbacks(app, cache)

if __name__ == "__main__":
    # Check if dashboard file exists, create if not
    dashboard_file = get_dashboard_file_path()

    if not os.path.exists(dashboard_file):
        print(f"Dashboard file {dashboard_file} does not exist. Creating it...")
        subprocess.run(["python", "create_dash.py"])
    else:
        print(f"Dashboard file {dashboard_file} already exists. Using existing file.")

    # Get server settings from config
    server_config = config["app_settings"]["server"]
    host = server_config["host"]
    port = server_config["port"]
```

```
    debug = config["app_settings"]["debug_mode"]

    # Run the server
    app.run_server(debug=debug, host=host, port=port)
```

4 config$_a$nd$_d$ata.py

```
"""
Configuration and Data Loading Module

This module handles:
1. Loading the configuration from JSON file
2. Loading static content like markdown files
3. Loading base data from Excel files
4. Preparing dropdown options and other static data structures
"""

import os
import json
import pandas as pd
from datetime import datetime


def load_config():
    """
    Load the configuration from the JSON file.

    Returns:
        dict: Configuration dictionary
    """
    with open("config.json", "r") as f:
        return json.load(f)


# Load configuration
config = load_config()


def load_markdown_content():
    """
    Load markdown content from files.

    Returns:
        tuple: (institute_view_text, fru_text)
```

```
    """
    # Load Institute view markdown
    md_path = os.path.join("assets", "Institute_View.md")
    with open(md_path, "r", encoding="utf-8") as f:
        institute_view_text = f.read()

    # Load FRV markdown
    frv_path = os.path.join("assets", "FRV.md")
    with open(frv_path, "r", encoding="utf-8") as f:
        frv_text = f.read()

    return institute_view_text, frv_text


# Load markdown content
institute_view_text, frv_text = load_markdown_content()


# Extract metric configuration from config file
mapping_metric = config["metrics"]["mapping"]
metric_labels = config["metrics"]["labels"]
metric_colors = config["metrics"]["colors"]
metric_descriptions = config["metrics"]["descriptions"]

def load_base_data():
    """
    Load the base data from the MacroEdge Excel file.

    Returns:
        tuple: (df_names, df_variables, df_index) DataFrames
    """
    file_paths = config["file_paths"]
    macroedge_file = file_paths["macroedge_file"]

    df_names = pd.read_excel(macroedge_file, sheet_name=file_paths["names_sheet"])
    df_variables = pd.read_excel(macroedge_file, sheet_name=file_paths["variables_sheet"])
    df_index = pd.read_excel(macroedge_file, sheet_name=file_paths["indexdata_sheet"])

    return df_names, df_variables, df_index


# Load the data once at module import time
df_names, df_variables, df_index = load_base_data()


def build_dropdown_options():
```

```
    """
    Build dropdown options for countries and metrics.

    Returns:
        tuple: (country_options, metric_options)
    """
    # Build country options
    country_options = [{"label": "All", "value": "All"}] + [
        {"label": name, "value": name} for name in df_names["Country"].dropna().unique()
    ]

    # Build metric options (for modal dropdown)
    metric_options = (
        [{"label": "All", "value": "All"}] +
        [{"label": str(x), "value": str(x)} for x in sorted(df_variables["F/R/V"].dropna().u
    )

    return country_options, metric_options


# Build dropdown options
country_options, metric_options = build_dropdown_options()


# Extract dropdown texts from configuration
dropdown_texts = config["dropdown_texts"]


def get_dashboard_file_path():
    """
    Constructs the path to the current month's dashboard file.

    Returns:
        str: Path to the dashboard file
    """
    month_format = config["data_processing"]["date_format"]["month_pattern"]
    month_name = datetime.now().strftime(month_format)
    folder_path = os.path.dirname(os.path.abspath(config["file_paths"]["macroedge_file"])) +
    dashboard_file = folder_path + config["file_paths"]["dashboard_file_pattern"].replace("+
    return dashboard_file


def load_dashboard_data():
    """
    Loads the dashboard data from the current month's file.
```

```
    Returns:
        pandas.DataFrame: Dashboard data or None if file not found
    """
    dashboard_file = get_dashboard_file_path()
    if not os.path.exists(dashboard_file):
        return None

    return pd.read_excel(dashboard_file, sheet_name="Dashboard")


def get_valid_column_sets():
    """
    Get sets of valid columns for each metric category.

    Returns:
        tuple: (valid_fund, valid_risk, valid_val) Sets of column names
    """
    valid_fund = set(df_variables[df_variables["F/R/V"].str.lower() == "fundamentals"]["What
    valid_risk = set(df_variables[df_variables["F/R/V"].str.lower() == "risks"]["What"].str.
    valid_val = set(df_variables[df_variables["F/R/V"].str.lower() == "valuations"]["What"].

    return valid_fund, valid_risk, valid_val


# Get valid column sets
valid_fund, valid_risk, valid_val = get_valid_column_setss()
```

5 layout.py

```
"""
Main Layout Module

This module defines the Dash application layout components including:
- App initialization
- Header components
- Sidebar components
- Main content containers
- Modal dialogs
"""

import dash
from dash import html, dcc
import dash_bootstrap_components as dbc
```

```python
from dash_table.Format import Format, Scheme
import pandas as pd

# Import configuration and data
from .config_and_data import (
    config,
    country_options,
    dropdown_texts,
    metric_options,
    metric_descriptions
)

# Get style settings from config
style_config = config["app_settings"]["style"]

# Initialize the Dash app
external_stylesheets = [
    getattr(dbc.themes, style_config["theme"]),
    style_config["icons_url"]
]

app = dash.Dash(
    __name__,
    external_stylesheets=external_stylesheets,
    assets_folder="assets",
    suppress_callback_exceptions=True
)

# ====================
# Layout Components
# ====================

# Amundi logo image
fixed_image = html.Img(
    src=app.get_asset_url("Amundi.jpg"),
    style={
        "position": "relative",
        "top": "10px",
        "right": "10px",
        "height": style_config["header_height"]
    }
)

# Header row with logo, title, and navigation buttons
header_row = dbc.Row(
    [
```

```
        dbc.Col(fixed_image, width="auto"),
        dbc.Col(
            html.H1(config["app_settings"]["header_title"], className="text-center"),
            className="d-flex align-items-center justify-content-center",
        ),
        dbc.Col(
            dbc.ButtonGroup(
                [
                    dbc.Button(
                        dropdown_texts["btn_actual"],
                        id="btn-actual",
                        style={"background-color": style_config["primary_color"], "color": '
                        className="m-1"
                    ),
                    dbc.Button(
                        dropdown_texts["btn_historical"],
                        id="btn-historical",
                        style={"background-color": style_config["secondary_color"], "color"
                        className="m-1"
                    ),
                    dbc.Button(
                        dropdown_texts["btn_about"],
                        id="btn-about",
                        style={"background-color": style_config["tertiary_color"], "color":
                        className="m-1"
                    ),
                ]
            ),
            width="auto",
            className="d-flex align-items-center justify-content-end",
        ),
    ],
    align="center",
    className="mb-3"
)

# Title row
title_row = dbc.Row(
    dbc.Col(
        html.H1(config["app_settings"]["header_title"], className="text-center mt-3 mb-4")
    )
)

# Right sidebar with country selection and data
country_info_style = config["layout"]["country_info"]
accordion_config = config["layout"]["accordion"]
```

```
right_sidebar = dbc.Col(
    [
        html.Label(
            dropdown_texts["dropdown_countries_label"],
            style={"fontWeight": "bold"}
        ),
        dcc.Dropdown(
            id="dropdown-country",
            options=country_options,
            placeholder="Select a country...",
            value="All",
            style={"width": "100%"}
        ),
        html.Br(),
        html.Div(
            [
                html.Img(
                    id="country-flag",
                    style={"height": country_info_style["flag_height"],
                           "margin-right": country_info_style["margin_right"]}
                ),
                html.P(id="country-weight", className="mt-2 mb-0"),
                html.P(id="country-sectors", className="mt-0"),
                html.P(id="country-scores", className="mt-0")
            ],
            style={"textAlign": country_info_style["text_align"]}
        ),
        html.Hr(),
        dbc.Accordion(
            [
                dbc.AccordionItem(
                    [html.P("", id="pop-data")],
                    title="Country Data",
                    id="country-data-accordion",
                    item_id="country_data"
                ),
                dbc.AccordionItem(
                    [html.Div(id="economic-table")],
                    title="Economic Data",
                    id="economic-data-accordion",
                    item_id="economic"
                ),
                dbc.AccordionItem(
                    [html.Div(id="forecast-table")],
                    title="Institute Forecast",
```

```
                    id="forecast-accordion",
                    item_id="forecast"
                ),
                dbc.AccordionItem(
                    html.Div(id="financial-data-container"),
                    title="Financial Data",
                    item_id="financial"
                )
            ],
            always_open=accordion_config["always_open"],
            active_item=accordion_config["default_active"]
        ),
    ],
    id="right-sidebar",
    width=config["layout"]["sidebar"]["width"],
    className=config["layout"]["sidebar"]["className"],
)

# Store components for selected Metric and Scenario
store_components = [
    dcc.Store(id="selected-metric", data="All"),
    dcc.Store(id="selected-scenario", data="standard")
]

# Row with clickable links for Metric and Scenario selection
selection_row = dbc.Row(
    [
        dbc.Col(
            html.H4("Macro Positioning", className="mb-0"),
            width="auto"
        ),
        dbc.Col(
            dbc.Row(
                [
                    dbc.Col(
                        [
                            html.Label("Metric:", className="mb-0 me-2"),
                            dbc.Button(
                                "All",
                                id="metric-link",
                                n_clicks=0,
                                color="link",
                                style={"padding": "0", "fontWeight": "normal"}
                            )
                        ],
                        width="auto",
```

```
                        className="d-flex align-items-center"
                    ),
                    dbc.Col(
                        [
                            html.Label("Scenario:", className="mb-0 me-2"),
                            dbc.Button(
                                "Standard",
                                id="scenario-link",
                                n_clicks=0,
                                color="link",
                                style={"padding": "0", "fontWeight": "normal"}
                            )
                        ],
                        width="auto",
                        className="d-flex align-items-center"
                    ),
                    dbc.Col(
                        dbc.Button(
                            "Extract Data",
                            id="extract-data-button",
                            n_clicks=0,
                            color="link",
                            style={"padding": "0", "fontWeight": "normal"}
                        ),
                        width="auto",
                        className="d-flex align-items-center"
                    )
                ],
                className="g-2"
            ),
            width=True,
            className="d-flex justify-content-end align-items-center"
        )
    ],
    className="mb-3",
    id="selection-row"  # Aggiungi un ID al selection_row
)

# Modifica il about_content per includere il selection_row ma nasconderlo
about_content = html.Div(
    [
        # Includi il selection_row ma con stile display:none
        html.Div(
            selection_row,
            style={"display": "none"}
        ),
```

```
        dcc.Markdown(id="institute-view-content",
                     children="",
                     style={"padding": "20px"})
    ]
)

# Modifica il historical_content per includere il selection_row ma nasconderlo
historical_content = html.Div(
    [
        # Includi il selection_row ma con stile display:none
        html.Div(
            selection_row,
            style={"display": "none"}
        ),
        dcc.Markdown(id="frv-content",
                     children="",
                     style={"padding": "20px"})
    ]
)

# Modifica il actual_content per rendere esplicito il selection_row
actual_content = dbc.Container([
    selection_row,
    dbc.Row([
         dbc.Col(
             dcc.Graph(id="polar-plot", style={"height": "400px", "width": "100%"}),
             width=6,
             id="polar-plot-col"
         ),
         dbc.Col(
             dcc.Graph(id="bar-plot", style={"height": "400px", "width": "100%"}),
             width=6,
             id="bar-plot-col"
         )
    ]),
    dbc.Row([
         dbc.Col(html.Div(id="dashboard-table-container"), width=12)
    ]),
    dbc.Col(html.Hr(), width=12),
    dbc.Row([dbc.Col(html.H4("Index overview", className="text-left"), width=12)]),
    dbc.Row([dbc.Col(html.H5("Total Return Decomposition", className="text-left"), width=12)
    dbc.Row([dbc.Col(html.Div(id="total-return-decomposition"), width=12)]),
    dbc.Col(html.Hr(), width=12),
    dbc.Row([dbc.Col(html.Div(id="flow-analysis-section"), width=12)]),
])
```

```
# Modal for Metric selection
metric_modal = dbc.Modal(
    [
        dbc.ModalHeader(dropdown_texts["modal_metric_title"]),
        dbc.ModalBody([
            dbc.ListGroup([
                dbc.ListGroupItem(
                    [
                        html.Div("All", style={"fontWeight": "bold"}),
                        html.Div(metric_descriptions["All"], style={"fontSize": "smaller", '
                    ],
                    id="metric-option-all", action=True
                ),
                dbc.ListGroupItem(
                    [
                        html.Div("Fundamentals", style={"fontWeight": "bold"}),
                        html.Div(metric_descriptions["Fundamentals"], style={"fontSize": "sm
                    ],
                    id="metric-option-fundamentals", action=True
                ),
                dbc.ListGroupItem(
                    [
                        html.Div("Risks", style={"fontWeight": "bold"}),
                        html.Div(metric_descriptions["Risks"], style={"fontSize": "smaller",
                    ],
                    id="metric-option-risks", action=True
                ),
                dbc.ListGroupItem(
                    [
                        html.Div("Valuations", style={"fontWeight": "bold"}),
                        html.Div(metric_descriptions["Valuations"], style={"fontSize": "smal
                    ],
                    id="metric-option-valuations", action=True
                )
            ])
        ]),
        dbc.ModalFooter(dbc.Button("Close", id="close-metric-modal", className="ml-auto"))
    ],
    id="metric-modal",
    is_open=False
)

# Modal for Scenario selection
scenario_modal = dbc.Modal(
    [
        dbc.ModalHeader(
```

```
                [
                    html.Span(dropdown_texts["modal_scenario_title"]),
                    dbc.Button(
                        html.I(className="bi bi-info-circle"),
                        id="scenario-info-btn",
                        color="link",
                        style={"marginLeft": "10px"}
                    )
                ]
            ),
            dbc.ModalBody([
                dbc.ListGroup([
                    dbc.ListGroupItem(
                        [
                            html.Div(scenario["name"], style={"fontWeight": "bold"}),
                            html.Div(scenario["description"], style={"fontSize": "smaller", "col
                        ],
                        id=f"scenario-option-{scenario['id']}", action=True
                    )
                    for scenario in config["scenarios"]["list"]
                ])
            ]),
            dbc.ModalFooter(dbc.Button("Close", id="close-scenario-modal", className="ml-auto"))
        ],
        id="scenario-modal",
        is_open=False
    )

    # Modal for PPT extraction
    extract_modal = dbc.Modal(
        [
            dbc.ModalHeader("Extract Data"),
            dbc.ModalBody([
                html.Label("Select Countries:"),
                dcc.Checklist(
                    id="extract-country-checklist",
                    options=[
                        {"label": c, "value": c}
                        for c in ["All"] + sorted([opt["value"] for opt in country_options if op
                    ],
                    value=[],
                    labelStyle={"display": "block", "margin": "5px 0"}
                ),
                html.Br(),
                dbc.Button("Download PPT", id="download-ppt-button", color="primary")
            ]),
```

```
        dbc.ModalFooter(dbc.Button("Close", id="close-extract-modal", className="ml-auto"))
    ],
    id="extract-modal",
    is_open=False
)

# Download component for PPT download
download_component = dcc.Download(id="download-ppt")

# Assemble the complete app layout
app_layout = dbc.Container(
    store_components +
    [
        header_row,
        dbc.Row([
            dbc.Col(
                html.Div(id="content", children=actual_content),
                width=config["layout"]["content"]["width"]
            ),
            right_sidebar
        ]),
        extract_modal,
        download_component,
        metric_modal,
        scenario_modal
    ],
    fluid=config["layout"]["containers"]["main"]["fluid"],
)

# Attach the layout to the app
app.layout = app_layout


# ================================
# Functions for chart generation
# ================================

def make_stacked_bar_for_all_indices():
    """
    Create a stacked bar chart for all global indices.

    Returns:
        plotly.graph_objects.Figure: Stacked bar chart
    """
    import plotly.graph_objects as go
    from .utils import cagr
```

```
import math

skip_rows = config["file_paths"]["bloomberg_skiprows"]

# Labels and colors for chart components
stacked_config = config["charts"]["stacked_bar"]["variables"]
labels = {var["id"]: var["label"] for var in stacked_config}
colors = {var["id"]: var["color"] for var in stacked_config}

# Add dividend configuration
dividend_config = config["charts"]["stacked_bar"]["dividend_config"]
labels[dividend_config["id"]] = dividend_config["label"]
colors[dividend_config["id"]] = dividend_config["color"]

# List of global indices
index_list = config["charts"]["financial"]["stacked"]

# Time calculation constant
year_days = config["data_processing"]["calculations"]["year_days"]

# Data container for chart
data_attr = {}

# Process each index
for entry in index_list:
    label, sheet_name = entry["name"], entry["sheet_name"]

    try:
        # Load data for this index
        df = pd.read_excel(config["file_paths"]["bloomberg"], sheet_name=sheet_name, ski
    except Exception:
        data_attr[label] = None
        continue

    if df.empty:
        data_attr[label] = None
        continue

    # Get date column
    date_col = df.columns[0]

    # Required columns for calculations
    required_cols = ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN",
                     "INDX_ADJ_PE", "PX_LAST", "TOTAL_RETURN"]

    if any(col not in df.columns for col in required_cols):
```

```
        data_attr[label] = None
        continue

    # Filter to required columns and remove rows with missing dates
    df = df[[date_col] + required_cols].dropna(subset=[date_col])

    if len(df) < 2:
        data_attr[label] = None
        continue

    # Get first and last rows for calculations
    first_row = df.iloc[0]
    last_row = df.iloc[-1]

    # Calculate time difference
    if isinstance(first_row[date_col], pd.Timestamp) and isinstance(last_row[date_col],
        days = (last_row[date_col] - first_row[date_col]).days
    else:
        days = 0

    year_frac = days / year_days if days > 0 else 1.0

    # Calculate CAGR for each component
    annual_sales = cagr(first_row["TRAIL_12M_SALES_PER_SH"], last_row["TRAIL_12M_SALES_P
    annual_margin = cagr(first_row["TRAIL_12M_PROF_MARGIN"], last_row["TRAIL_12M_PROF_MA
    annual_pe = cagr(first_row["INDX_ADJ_PE"], last_row["INDX_ADJ_PE"], year_frac)
    annual_px_last = cagr(first_row["PX_LAST"], last_row["PX_LAST"], year_frac)
    annual_total = cagr(first_row["TOTAL_RETURN"], last_row["TOTAL_RETURN"], year_frac)

    # Calculate log returns
    log_sales = math.log(1 + annual_sales) if annual_sales is not None else 0
    log_margin = math.log(1 + annual_margin) if annual_margin is not None else 0
    log_pe = math.log(1 + annual_pe) if annual_pe is not None else 0
    log_px_last = math.log(1 + annual_px_last) if annual_px_last is not None else 0
    log_total = math.log(1 + annual_total) if annual_total is not None else 0

    # For dividends, use difference between total return and price
    log_dividends = (annual_total - annual_px_last) if (annual_total is not None and ann

    # Calculate ratio for rebasing
    ratio = (annual_total / log_total) if (log_total != 0) else 0

    # Calculate rebased annualized return for each component
    rebased_sales = log_sales * ratio
    rebased_margin = log_margin * ratio
    rebased_pe = log_pe * ratio
```

```
        rebased_dividends = log_dividends * ratio

        # Store rebased values for this index
        comp_rebased = {
            "TRAIL_12M_SALES_PER_SH": rebased_sales,
            "TRAIL_12M_PROF_MARGIN": rebased_margin,
            "INDX_ADJ_PE": rebased_pe,
            "DIVIDENDS": rebased_dividends
        }
        data_attr[label] = comp_rebased

    # Create stacked bar chart
    fig = go.Figure()
    x_indices = [entry["name"] for entry in index_list]
    components = ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN", "INDX_ADJ_PE", "DIVIDEN

    # Add trace for each component
    for comp in components:
        y_vals = []
        for idx in x_indices:
            if data_attr.get(idx):
                y_vals.append(data_attr[idx][comp] * 100)  # Convert to percentage
            else:
                y_vals.append(0)

        fig.add_trace(go.Bar(
            x=x_indices,
            y=y_vals,
            name=labels.get(comp, comp),
            marker_color=colors.get(comp, "gray")
        ))

    # Get chart layout settings
    common_layout = config["charts"]["layout"]["common"]
    default_title = config["charts"]["financial"]["default_titles"]["stacked_bar_all"]

    # Update layout
    fig.update_layout(
        barmode="stack",
        title=default_title,
        xaxis_title="Index",
        yaxis_title="Rebased Annualized Return (%)",
        plot_bgcolor=common_layout["plot_bgcolor"],
        legend=dict(
            orientation=common_layout["legend_position"]["orientation"],
            yanchor=common_layout["legend_position"]["yanchor"],
```

```
            y=common_layout["legend_position"]["y"],
            xanchor=common_layout["legend_position"]["xanchor"],
            x=common_layout["legend_position"]["x"]
        )
    )

    return dcc.Graph(figure=fig)


def make_stacked_bar_for_country(selected_country):
    """
    Create a stacked bar chart comparing a country to MXEF.

    Args:
        selected_country: Country to compare with MXEF

    Returns:
        plotly.graph_objects.Figure: Stacked bar chart
    """
    import plotly.graph_objects as go
    from .utils import cagr
    import math

    skip_rows = config["file_paths"]["bloomberg_skiprows"]

    # Labels and colors for chart components
    stacked_config = config["charts"]["stacked_bar"]["variables"]
    labels = {var["id"]: var["label"] for var in stacked_config}
    colors = {var["id"]: var["color"] for var in stacked_config}

    # Add dividend configuration
    dividend_config = config["charts"]["stacked_bar"]["dividend_config"]
    labels[dividend_config["id"]] = dividend_config["label"]
    colors[dividend_config["id"]] = dividend_config["color"]

    # Get sheet name for selected country
    mapping = config["countries"]["mapping"].get(selected_country, {})
    country_sheet = mapping.get("sheet_name", "")

    # Define indices to compare
    index_list = [
        {"name": "MXEF", "sheet_name": "MXEF Index"},
        {"name": selected_country, "sheet_name": country_sheet}
    ]

    # Time calculation constant
```

```
year_days = config["data_processing"]["calculations"]["year_days"]

# Data container for chart
data_attr = {}

# Process each index
for entry in index_list:
    label, sheet_name = entry["name"], entry["sheet_name"]

    try:
        # Load data for this index
        df = pd.read_excel(config["file_paths"]["bloomberg"], sheet_name=sheet_name, ski
    except Exception:
        data_attr[label] = None
        continue

    if df.empty:
        data_attr[label] = None
        continue

    # Get date column
    date_col = df.columns[0]

    # Required columns for calculations
    required_cols = ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN",
                     "INDX_ADJ_PE", "PX_LAST", "TOTAL_RETURN"]

    if any(col not in df.columns for col in required_cols):
        data_attr[label] = None
        continue

    # Filter to required columns and remove rows with missing dates
    df = df[[date_col] + required_cols].dropna(subset=[date_col])

    if len(df) < 2:
        data_attr[label] = None
        continue

    # Get first and last rows for calculations
    first_row = df.iloc[0]
    last_row = df.iloc[-1]

    # Calculate time difference
    if isinstance(first_row[date_col], pd.Timestamp) and isinstance(last_row[date_col],
        days = (last_row[date_col] - first_row[date_col]).days
    else:
```

```
            days = 0

        year_frac = days / year_days if days > 0 else 1.0

        # Calculate CAGR for each component
        annual_sales = cagr(first_row["TRAIL_12M_SALES_PER_SH"], last_row["TRAIL_12M_SALES_P
        annual_margin = cagr(first_row["TRAIL_12M_PROF_MARGIN"], last_row["TRAIL_12M_PROF_MA
        annual_pe = cagr(first_row["INDX_ADJ_PE"], last_row["INDX_ADJ_PE"], year_frac)
        annual_px_last = cagr(first_row["PX_LAST"], last_row["PX_LAST"], year_frac)
        annual_total = cagr(first_row["TOTAL_RETURN"], last_row["TOTAL_RETURN"], year_frac)

        # Calculate log returns
        log_sales = math.log(1 + annual_sales) if annual_sales is not None else 0
        log_margin = math.log(1 + annual_margin) if annual_margin is not None else 0
        log_pe = math.log(1 + annual_pe) if annual_pe is not None else 0
        log_px_last = math.log(1 + annual_px_last) if annual_px_last is not None else 0
        log_total = math.log(1 + annual_total) if annual_total is not None else 0

        # For dividends, use difference between total return and price
        log_dividends = (annual_total - annual_px_last) if (annual_total is not None and ann

        # Calculate ratio for rebasing
        ratio = (annual_total / log_total) if (log_total != 0) else 0

        # Calculate rebased annualized return for each component
        rebased_sales = log_sales * ratio
        rebased_margin = log_margin * ratio
        rebased_pe = log_pe * ratio
        rebased_dividends = log_dividends * ratio

        # Store rebased values for this index
        comp_rebased = {
            "TRAIL_12M_SALES_PER_SH": rebased_sales,
            "TRAIL_12M_PROF_MARGIN": rebased_margin,
            "INDX_ADJ_PE": rebased_pe,
            "DIVIDENDS": rebased_dividends
        }
        data_attr[label] = comp_rebased

# Create stacked bar chart
fig = go.Figure()
x_indices = [entry["name"] for entry in index_list]
components = ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN", "INDX_ADJ_PE", "DIVIDEN

# Add trace for each component
for comp in components:
```

```
            y_vals = []
            for idx in x_indices:
                if data_attr.get(idx):
                    y_vals.append(data_attr[idx][comp] * 100)  # Convert to percentage
                else:
                    y_vals.append(0)

            fig.add_trace(go.Bar(
                x=x_indices,
                y=y_vals,
                name=labels.get(comp, comp),
                marker_color=colors.get(comp, "gray")
            ))

        # Get chart layout settings
        common_layout = config["charts"]["layout"]["common"]
        default_title = config["charts"]["financial"]["default_titles"]["stacked_bar_country"].f

        # Update layout
        fig.update_layout(
            barmode="stack",
            title=default_title,
            xaxis_title="Index",
            yaxis_title="Rebased Annualized Return (%)",
            plot_bgcolor=common_layout["plot_bgcolor"],
            legend=dict(
                orientation=common_layout["legend_position"]["orientation"],
                yanchor=common_layout["legend_position"]["yanchor"],
                y=common_layout["legend_position"]["y"],
                xanchor=common_layout["legend_position"]["xanchor"],
                x=common_layout["legend_position"]["x"]
            )
        )

        return dcc.Graph(figure=fig)


    def make_total_return_for_country(selected_country):
        """
        Generate a table with total return decomposition for the selected country.

        Args:
            selected_country: Country to analyze

        Returns:
            dash_table.DataTable: Data table with return decomposition
```

```
"""
from dash import dash_table, html
from .utils import cagr, growth, format_value
import math

skip_rows = config["file_paths"]["bloomberg_skiprows"]
mapping = config["countries"]["mapping"].get(selected_country, {})
sheet_name = mapping.get("sheet_name", "")

if not sheet_name:
    return html.Div(f"No sheet found for {selected_country}")

try:
    df_bbg = pd.read_excel(config["file_paths"]["bloomberg"], sheet_name=sheet_name, ski
except Exception as e:
    return html.Div(f"Error reading data for {selected_country}: {str(e)}")

if df_bbg.empty:
    return html.Div(f"No data in sheet '{sheet_name}' for {selected_country}")

date_col = df_bbg.columns[0]

# Define columns in the desired order
col_list = ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN", "INDX_ADJ_PE", "DIVIDENDS

# Check for missing columns
missing = [c for c in col_list if c not in df_bbg.columns and c != "DIVIDENDS"]
if missing:
    return html.Div(f"Missing columns in {sheet_name}: {missing}")

# Filter and clean data
df_bbg = df_bbg[[date_col] + [c for c in col_list if c != "DIVIDENDS"]].dropna(subset=[c
if len(df_bbg) < 2:
    return html.Div("Not enough data to compute decomposition.")

first_row = df_bbg.iloc[0]
last_row = df_bbg.iloc[-1]

# Calculate time period
if isinstance(first_row[date_col], pd.Timestamp) and isinstance(last_row[date_col], pd.T
    days = (last_row[date_col] - first_row[date_col]).days
else:
    days = 0

# Get year days constant from config
year_days = config["data_processing"]["calculations"]["year_days"]
```

```
year_frac = days / year_days if days > 0 else 1

# Calculate growth and CAGR for original columns
growth_dic = {}
annual_dic = {}
for c in ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN", "INDX_ADJ_PE", "PX_LAST", '
    v0 = first_row[c]
    v1 = last_row[c]
    if c == "INDX_DIVISOR":
        growth_dic[c] = -growth(v0, v1)
        annual_dic[c] = -cagr(v0, v1, year_frac)
    else:
        growth_dic[c] = growth(v0, v1)
        annual_dic[c] = cagr(v0, v1, year_frac)

# Calculate DIVIDENDS column
growth_div = None
annual_div = None
if first_row["PX_LAST"] and first_row["TOTAL_RETURN"]:
    growth_div = growth(first_row["TOTAL_RETURN"], last_row["TOTAL_RETURN"]) - growth(fi
    annual_div = (1+cagr(first_row["TOTAL_RETURN"], last_row["TOTAL_RETURN"], year_frac)

growth_dic["DIVIDENDS"] = growth_div
annual_dic["DIVIDENDS"] = annual_div

# Calculate Return Attribution Annualized
cols_for_attribution = ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN", "INDX_ADJ_PE"
sum_annual_attr = sum(annual_dic[c] for c in cols_for_attribution if annual_dic[c] is no

# Get minimum denominator value from config
min_denom = config["data_processing"]["calculations"]["minimum_cagr_denom"]
if abs(sum_annual_attr) < min_denom:
    sum_annual_attr = min_denom

return_attr = {}
return_attr_norm = {}
for c in cols_for_attribution:
    val = annual_dic[c]
    if val is not None:
        return_attr[c] = (val * annual_dic["TOTAL_RETURN"]) / sum_annual_attr
        return_attr_norm[c] = return_attr[c] / annual_dic["TOTAL_RETURN"] if annual_dic[
    else:
        return_attr[c] = None
        return_attr_norm[c] = None

def fmt_val(x, pct=False):
```

```
    """Format a value for display."""
    if x is None:
        return "N/A"
    try:
        if pct:
            return f"{x * 100:.1f}%"
        else:
            return f"{x:.2f}"
    except:
        return str(x)

# Prepare table rows
rows_data = []

# Initial date row
r1 = {"Label": f"Initial date: {str(df_bbg.iloc[0][date_col])[:10]}"}

# Final date row
r2 = {"Label": f"Final date: {str(df_bbg.iloc[-1][date_col])[:10]}"}

# Growth percentage row
r3 = {"Label": "Growth (%)"}

# Annual Average (CAGR) row
r4 = {"Label": "Annual Avg (CAGR)"}

# Log Return row
r5 = {"Label": "Log Return (Annual Avg)"}

# Rebased Annualized Return row
r6 = {"Label": "Rebased Annualized Return"}

# Calculate log returns
log_row = {}
for c in col_list:
    if c in ["TRAIL_12M_SALES_PER_SH", "TRAIL_12M_PROF_MARGIN", "INDX_ADJ_PE", "PX_LAST"
        val = annual_dic[c]
        log_row[c] = math.log(1 + val) if val is not None else None
    elif c == "DIVIDENDS":
        # For DIVIDENDS, log return is the simple difference
        if annual_dic["TOTAL_RETURN"] is not None and annual_dic["PX_LAST"] is not None:
            log_row[c] = annual_dic["TOTAL_RETURN"] - annual_dic["PX_LAST"]
        else:
            log_row[c] = None

# Calculate ratio for rebasing
```

```
total_log = math.log(1 + annual_dic["TOTAL_RETURN"]) if annual_dic["TOTAL_RETURN"] is no
ratio = (annual_dic["TOTAL_RETURN"] / total_log) if (total_log not in [None, 0]) else No

# Calculate rebased annualized return
rebased_row = {}
for c in col_list:
    if log_row.get(c) is not None and ratio is not None:
        rebased_row[c] = log_row[c] * ratio
    else:
        rebased_row[c] = None

# Populate table rows
for c in col_list:
    r1[c] = fmt_val(first_row[c]) if c != "DIVIDENDS" else ""
    r2[c] = fmt_val(last_row[c]) if c != "DIVIDENDS" else ""
    r3[c] = fmt_val(growth_dic[c], pct=True)
    r4[c] = fmt_val(annual_dic[c], pct=True)
    r5[c] = fmt_val(log_row[c], pct=True)
    r6[c] = fmt_val(rebased_row[c], pct=True)

rows_data.extend([r1, r2, r3, r4, r5, r6])

# Define table columns
columns = [{"name": str("Label"), "id": str("Label")}] + [{"name": str(c), "id": str(c)]

# Get table styles from config
table_styles = config["tables"]["return_decomposition_table"]["styles"]

# Create data table
dt = dash_table.DataTable(
    data=rows_data,
    columns=columns,
    style_cell=table_styles["cell"],
    style_header=table_styles["header"],
    style_table=table_styles["table"],
    page_action="none"
)

return dt
```

6 utils.py

```python
"""
Utility Functions Module

This module contains helper functions that are used across the application.
"""

import math
import pandas as pd
from dash import html

# Import configuration
from .config_and_data import config


def cagr(first_val, last_val, year_frac):
    """
    Calculate Compound Annual Growth Rate (CAGR).

    Args:
        first_val: Initial value
        last_val: Final value
        year_frac: Time period in years

    Returns:
        float: CAGR value or None if calculation is not possible
    """
    if first_val in [None, 0] or year_frac <= 0:
        return None
    return (last_val / first_val) ** (1 / year_frac) - 1


def growth(first_val, last_val):
    """
    Calculate simple growth percentage.

    Args:
        first_val: Initial value
        last_val: Final value

    Returns:
        float: Growth percentage or None if calculation is not possible
    """
    if first_val is None or first_val == 0:
```

```
        return None
    return (last_val / first_val) - 1


def format_percentage(val, precision=None):
    """
    Format a value as a percentage string.

    Args:
        val: Value to format
        precision: Number of decimal places

    Returns:
        str: Formatted percentage string
    """
    if precision is None:
        precision = config["data_processing"]["calculations"]["numeric_precision"]["percenta

    try:
        return f"{float(val):.{precision}f}%"
    except (ValueError, TypeError):
        return str(val)


def lighten_color(color, factor=None):
    """
    Lightens an RGB color by interpolating towards white.

    Args:
        color: RGB color string in format "rgb(R, G, B)"
        factor: Interpolation factor (0 = original color, 1 = white)

    Returns:
        str: Lightened RGB color string
    """
    if factor is None:
        factor = config["charts"]["layout"]["bar"]["lighten_factor"]

    # Convert "rgb(R, G, B)" to a list of integers
    rgb = list(map(int, color.replace("rgb(", "").replace(")", "").split(",")))
    # Interpolate towards white (255, 255, 255)
    lighter = [int(c + (255 - c) * factor) for c in rgb]
    return f"rgb({lighter[0]}, {lighter[1]}, {lighter[2]})"


def get_color_for_score(score):
```

```
    """
    Determine color based on a score value.

    Args:
        score: Numeric score (0-100)

    Returns:
        str: Color name (red, green, gold, or black for None)
    """
    if score is None:
        return "black"

    thresholds = config["data_processing"]["calculations"]["score_thresholds"]

    if score > thresholds["high"]:
        return "green"
    elif score < thresholds["low"]:
        return "red"
    else:
        return "gold"


def create_score_display(label, score):
    """
    Create a colored score display component.

    Args:
        label: Label text
        score: Score value

    Returns:
        html.P: Dash HTML component with colored score
    """
    score_int = int(round(score)) if score is not None else "N/A"
    return html.P([
        html.Span(f"{label}: ", style={'color': 'black'}),
        html.Span(f"{score_int}", style={'color': get_color_for_score(score)})
    ], style={'textAlign': 'center'})


def filter_valid_columns(df, valid_cols_set):
    """
    Filter a DataFrame to keep only columns that are in the valid set.

    Args:
        df: DataFrame to filter
```

```
        valid_cols_set: Set of valid column names

    Returns:
        list: List of column names that are in both the DataFrame and valid set
    """
    return [col for col in df.columns if col in valid_cols_set]


def order_dataframe_by_country(df, country):
    """
    Reorders a DataFrame to put a specific country first.

    Args:
        df: DataFrame to reorder
        country: Country to prioritize

    Returns:
        pandas.DataFrame: Reordered DataFrame
    """
    df_country = df[df["Country"] == country]
    df_other = df[df["Country"] != country]
    return pd.concat([df_country, df_other])


def format_value(value, is_percentage=False, precision=None):
    """
    Format a value for display, handling None values.

    Args:
        value: Value to format
        is_percentage: Whether to format as percentage
        precision: Number of decimal places

    Returns:
        str: Formatted value
    """
    if value is None:
        return "N/A"

    if precision is None:
        if is_percentage:
            precision = config["data_processing"]["calculations"]["numeric_precision"]["perc
        else:
            precision = config["data_processing"]["calculations"]["numeric_precision"]["deci

    try:
```

```
        if is_percentage:
            return f"{value * 100:.{precision}f}%"
        else:
            return f"{value:.{precision}f}"
    except (ValueError, TypeError):
        return str(value)


def format_with_commas(value):
    """
    Format a numeric value with commas for thousands.

    Args:
        value: Numeric value to format

    Returns:
        str: Formatted value with commas
    """
    try:
        return f"{int(value):,}"
    except (ValueError, TypeError):
        return str(value)


def apply_scenario_weights(df, scenario):
    """
    Apply scenario weights to calculate total score.

    Args:
        df: DataFrame with base scores
        scenario: Scenario name

    Returns:
        pandas.DataFrame: DataFrame with updated total score
    """
    from .config_and_data import valid_fund, valid_risk, valid_val

    # Use standard scenario if none specified
    if scenario is None or scenario not in config["scenarios"]["weights"]:
        scenario = "standard"

    # Get weights from configuration
    weights = config["scenarios"]["weights"][scenario]

    # Calculate scores based on column means
    df["Fundamental Score"] = df[[col for col in df.columns if col in valid_fund]].mean(axis
```

```
    df["Risk Score"] = df[[col for col in df.columns if col in valid_risk]].mean(axis=1)
    df["Valuation Score"] = df[[col for col in df.columns if col in valid_val]].mean(axis=1)

    # Apply weighted total score
    df["TotalScore"] = (
        df["Fundamental Score"] * (weights["fundamental"] / 100) +
        df["Risk Score"] * (weights["risk"] / 100) +
        df["Valuation Score"] * (weights["valuation"] / 100)
    )

    return df
```

7 Callbacks

7.1 init.py

```
"""
Callbacks Initialization Module

This module imports and registers all callback functions from the various callback modules.
"""

def register_callbacks(app, cache):
    """
    Register all callbacks with the Dash application.

    Args:
        app: The Dash application instance
        cache: The Flask-Cache instance for caching expensive operations
    """
    # Import all callback modules
    from .callback_modals import register_modal_callbacks
    from .callback_plots import register_plot_callbacks
    from .callback_tables import register_table_callbacks
    from .callback_ppt import register_ppt_callbacks

    # Register callbacks from each module
    register_modal_callbacks(app, cache)
    register_plot_callbacks(app, cache)
    register_table_callbacks(app, cache)
    register_ppt_callbacks(app, cache)

    print("All callbacks registered successfully.")
```

7.2 Tables.py

```
"""
Table and Data Display Callbacks Module

This module contains all callbacks related to tables and data displays:
- Dashboard table
- Economic data table
- Forecast data table
- Country data display
"""

import dash
from dash import Input, Output, html, dash_table
from dash.dependencies import State
from dash_table.Format import Format, Scheme
import pandas as pd
import os
from datetime import datetime
import dash_bootstrap_components as dbc
import dash_table

# Import configuration and utilities
from ..config_and_data import (
    config,
    df_variables,
    df_index,
    get_dashboard_file_path,
    valid_fund,
    valid_risk,
    valid_val
)

from ..utils import (
    apply_scenario_weights,
    order_dataframe_by_country,
    format_percentage,
    format_with_commas
)


def register_table_callbacks(app, cache):
    """
    Register all table-related callbacks with the app.

    Args:
```

```
    app: The Dash application instance
    cache: The Flask-Cache instance
"""

# Callback to update the Dashboard table
@app.callback(
    Output("dashboard-table-container", "children"),
    [Input("btn-actual", "n_clicks"),
     Input("dropdown-country", "value"),
     Input("selected-metric", "data"),
     Input("selected-scenario", "data")]
)
@cache.memoize(timeout=config["app_settings"]["cache"]["thresholds"]["tables"])
def update_dashboard_table(btn_actual, selected_country, selected_metric, scenario):
    """
    Update the main dashboard table with filtered and formatted data.

    Returns:
        dash.html.Div: Table component
    """
    dashboard_file = get_dashboard_file_path()

    if not os.path.exists(dashboard_file):
        return html.Div(config["messages"]["errors"]["dashboard_file_not_found"])

    df_full = pd.read_excel(dashboard_file, sheet_name="Dashboard")
    df_small = df_full.copy()
    df_main = df_full.copy()

    # Use standard scenario if none specified
    if scenario is None:
        scenario = "standard"

    # Apply scenario weights if not standard
    if scenario != "standard":
        df_small = apply_scenario_weights(df_small, scenario)
        df_main = apply_scenario_weights(df_main, scenario)

    # Filter for specific metric if requested
    if selected_metric and selected_metric != "All":
        valid_vars = df_variables[df_variables["F/R/V"] == selected_metric]["What"].unic
        columns_to_keep = ["Country"] + [col for col in df_main.columns if col in valid_
        # Always include score columns
        for score in ["Fundamental Score", "Risk Score", "Valuation Score", "TotalScore'
            if score not in columns_to_keep:
                columns_to_keep.append(score)
```

```
    df_main = df_main[columns_to_keep]

# Prioritize selected country if specified
if selected_country and selected_country != "All":
    df_main = order_dataframe_by_country(df_main, selected_country)
    df_small = order_dataframe_by_country(df_small, selected_country)

# Reorder columns for better display
desired_order = ["Country", "TotalScore", "Fundamental Score", "Valuation Score", "P
new_order = [col for col in desired_order if col in df_main.columns]
new_order += [col for col in df_main.columns if col not in new_order]
df_main = df_main[new_order]

# Get table styling configuration
table_styles = config["tables"]["dashboard_table"]["styles"]
quantile_thresholds = config["metrics"]["quantile_thresholds"]
color_ranges = config["metrics"]["color_ranges"]

# Define conditional styling for numeric data
style_data_conditional = []
numeric_cols = [c for c in df_main.columns if c != "Country" and pd.api.types.is_num

for col in numeric_cols:
    lower_bound = df_main[col].quantile(quantile_thresholds["lower"])
    upper_bound = df_main[col].quantile(quantile_thresholds["upper"])

    # Style for low values
    style_data_conditional.append({
        'if': {'filter_query': f'{{{col}}} <= {lower_bound}', 'column_id': col},
        'backgroundColor': color_ranges["low"]["background"],
        'color': color_ranges["low"]["text"]
    })

    # Style for high values
    style_data_conditional.append({
        'if': {'filter_query': f'{{{col}}} >= {upper_bound}', 'column_id': col},
        'backgroundColor': color_ranges["high"]["background"],
        'color': color_ranges["high"]["text"]
    })

# Style headers based on metric category
header_styles = []
for col in df_main.columns:
    # For aggregate score columns
    if col == config["metrics"]["mapping"].get("Fundamentals"):
        header_styles.append({
```

```
                "if": {"column_id": col},
                "backgroundColor": config["metrics"]["colors"]["Fundamentals"],
                "color": "white"
            })
        elif col == config["metrics"]["mapping"].get("Risks"):
            header_styles.append({
                "if": {"column_id": col},
                "backgroundColor": config["metrics"]["colors"]["Risks"],
                "color": "white"
            })
        elif col == config["metrics"]["mapping"].get("Valuations"):
            header_styles.append({
                "if": {"column_id": col},
                "backgroundColor": config["metrics"]["colors"]["Valuations"],
                "color": "white"
            })
        else:
            # For individual metric columns
            if col in valid_fund:
                header_styles.append({
                    "if": {"column_id": col},
                    "backgroundColor": config["metrics"]["colors"]["Fundamentals"],
                    "color": "white"
                })
            elif col in valid_risk:
                header_styles.append({
                    "if": {"column_id": col},
                    "backgroundColor": config["metrics"]["colors"]["Risks"],
                    "color": "white"
                })
            elif col in valid_val:
                header_styles.append({
                    "if": {"column_id": col},
                    "backgroundColor": config["metrics"]["colors"]["Valuations"],
                    "color": "white"
                })

    # Configure columns for the DataTable
    main_columns = []
    for col in df_main.columns:
        if pd.api.types.is_numeric_dtype(df_main[col]):
            main_columns.append({
                "name": str(col),
                "id": col,
                "type": "numeric",
                "format": Format(precision=0, scheme=Scheme.fixed)
```

```
            })
        else:
            main_columns.append({"name": str(col), "id": str(col)})

    # Get table settings from config
    page_size = config["tables"]["dashboard_table"]["default_page_size"]

    # Create the main table component
    main_table = dash_table.DataTable(
        data=df_main.to_dict("records"),
        columns=main_columns,
        page_size=page_size,
        style_table=table_styles["table"],
        style_cell=table_styles["cell"],
        style_header=table_styles["header"],
        fixed_columns={'headers': True, 'data': 1},
        sort_action="native",
        filter_action="none",
        style_data_conditional=style_data_conditional,
        style_header_conditional=header_styles
    )

    # For "All" country selection, show Leaders and Laggards tables
    if selected_country in (None, "All"):
        # Get summary table configuration
        summary_tables_config = config["tables"]["dashboard_table"]["summary_tables"]
        summary_styles = summary_tables_config["styles"]
        num_rows = summary_tables_config["num_rows"]

        base_cols = ["Country", "Fundamental Score", "Risk Score", "Valuation Score", "T
        for col in base_cols:
            if col not in df_small.columns:
                df_small[col] = "N/A"

        # Determine which column to use for ranking
        ranking_col = config["metrics"]["mapping"].get(selected_metric, "TotalScore") if
        remaining_cols = [col for col in base_cols if col not in ("Country", ranking_col
        small_cols_order = ["Country", ranking_col] + remaining_cols

        # Configure columns for the small tables
        small_columns = []
        for col in small_cols_order:
            if pd.api.types.is_numeric_dtype(df_small[col]):
                small_columns.append({
                    "name": str(col),
                    "id": col,
```

```
            "type": "numeric",
            "format": Format(precision=0, scheme=Scheme.fixed)
        })
    else:
        small_columns.append({"name": str(col), "id": str(col)})

# Sort for leaders (high scores) and laggards (low scores)
df_sorted_desc = df_small.sort_values(by=ranking_col, ascending=False)
df_sorted_asc = df_small.sort_values(by=ranking_col, ascending=True)
df_best = df_sorted_desc.head(num_rows)[small_cols_order]
df_worst = df_sorted_asc.head(num_rows)[small_cols_order]

# Create Leaders table
best_table = dash_table.DataTable(
    data=df_best.to_dict("records"),
    columns=small_columns,
    style_table=summary_styles["table"],
    style_cell=summary_styles["cell"],
    style_header=summary_styles["header"],
    page_action="none",
    style_as_list_view=True
)

# Create Laggards table
worst_table = dash_table.DataTable(
    data=df_worst.to_dict("records"),
    columns=small_columns,
    style_table=summary_styles["table"],
    style_cell=summary_styles["cell"],
    style_header=summary_styles["header"],
    page_action="none",
    style_as_list_view=True
)

# Combine Leaders and Laggards tables in a row
small_tables = dash.html.Div([
    dbc.Row([
        dbc.Col([
            html.H5(summary_tables_config["title_leaders"], className="text-cent
            best_table
        ], width=6),
        dbc.Col([
            html.H5(summary_tables_config["title_laggards"], className="text-cen
            worst_table
        ], width=6)
    ], className="mb-4")
```

```
        ])

        # Return Leaders, Laggards, and main table
        return html.Div([small_tables, main_table])
    else:
        # For specific country, just return the main table
        return main_table

# Callback to update Country Data
@app.callback(
    Output("pop-data", "children"),
    [Input("dropdown-country", "value")]
)
def update_country_data(selected_country):
    """
    Update the country demographic data display in the sidebar.

    Returns:
        dash.html.Div: Formatted country data
    """
    if not selected_country or selected_country == "All":
        return ""

    row = df_index[df_index["Country"] == selected_country]
    if row.empty:
        return html.Div("Population: N/A")

    row = row.iloc[0]
    pop = row.get("Population", "N/A")
    ppa = row.get("PPA", "N/A")
    gdp = row.get("GDP", "N/A")
    ltg = row.get("LTG", "N/A")

    # Format population with commas
    pop_formatted = format_with_commas(pop)

    return html.Div([
        html.P(f"Population: {pop_formatted}"),
        html.P(f"PPA: {ppa}"),
        html.P(f"GDP: {gdp}"),
        html.P(f"LTG: {ltg:.2}" if isinstance(ltg, (int, float)) else f"LTG: {ltg}")
    ])

# Callback to update Economic Data table
@app.callback(
    Output("economic-table", "children"),
```

```
    [Input("dropdown-country", "value")]
)
def update_economic_data(selected_country):
    """
    Update the economic data table for the selected country.

    Returns:
        dash.html.Table: Economic data table
    """
    if not selected_country or selected_country == "All":
        return ""

    row = df_index[df_index["Country"] == selected_country]
    if row.empty:
        return ""

    # Get economic table configuration
    economic_config = config["tables"]["economic_table"]
    headers = economic_config["headers"]
    indicators = economic_config["indicators"]
    styles = economic_config["styles"]

    # Create header row
    header_row_elem = html.Tr([
        html.Th(
            col,
            style={
                "backgroundColor": styles["header"]["background"],
                "color": styles["header"]["color"],
                "padding": styles["header"]["padding"]
            }
        ) for col in headers
    ])

    # Create data rows
    data_rows = []
    row_data = row.iloc[0]
    for indicator in indicators:
        indicator_name = indicator["name"]
        cols = indicator["columns"]

        row_cells = [html.Td(indicator_name, style={"padding": styles["cell"]["padding"]
        for col in cols:
            value = format_percentage(row_data.get(col, "N/A"))
            row_cells.append(html.Td(value, style={"padding": styles["cell"]["padding"]]
```

```
            data_rows.append(html.Tr(row_cells))

        # Create the table
        table = html.Table(
            [header_row_elem] + data_rows,
            style=styles["table"]
        )

        return table

    # Callback to update Forecast Data table
    @app.callback(
        Output("forecast-table", "children"),
        [Input("dropdown-country", "value")]
    )
    def update_forecast_data(selected_country):
        """
        Update the forecast data tables for the selected country.

        Returns:
            dash.html.Div: Container with forecast tables
        """
        if not selected_country or selected_country == "All":
            return ""

        row = df_index[df_index["Country"] == selected_country]
        if row.empty:
            return ""

        row_data = row.iloc[0]

        # --- Institute Forecast table ---
        forecast_config = config["tables"]["forecast_table"]
        fore_headers = forecast_config["headers"]
        fore_indicators = forecast_config["indicators"]
        fore_styles = forecast_config["styles"]

        # Create header row
        fore_header_row = html.Tr([
            html.Th(
                col,
                style={
                    "backgroundColor": fore_styles["header"]["background"],
                    "color": fore_styles["header"]["color"],
                    "padding": fore_styles["header"]["padding"]
                }
```

```
    )
    for col in fore_headers
])

# Create data rows
data_rows = []
for indicator in fore_indicators:
    indicator_name = indicator["name"]
    cols = indicator["columns"]

    row_cells = [html.Td(indicator_name, style={"padding": fore_styles["cell"]["pado
    for col in cols:
        value = format_percentage(row_data.get(col, "N/A"))
        row_cells.append(html.Td(value, style={"padding": fore_styles["cell"]["paddi

    data_rows.append(html.Tr(row_cells))

# Create the forecast table
forecast_table = html.Table(
    [fore_header_row] + data_rows,
    style=fore_styles["table"]
)

# --- GDP Growth and Inflation table ---
index_forecast_config = config["tables"]["index_forecast_table"]
index_headers = index_forecast_config["headers"]
index_indicators = index_forecast_config["indicators"]
index_styles = index_forecast_config["styles"]

indicator_header = html.Tr([
    html.Th(
        col,
        style={
            "backgroundColor": index_styles["header"]["background"],
            "color": index_styles["header"]["color"],
            "padding": index_styles["header"]["padding"]
        }
    ) for col in index_headers
])

# Create rows for each indicator
index_rows = []
for indicator in index_indicators:
    indicator_name = indicator["name"]
    cols = indicator["columns"]
```

```
            cells = [html.Td(indicator_name, style={"padding": index_styles["cell"]["padding
            for col in cols:
                value = format_percentage(row_data.get(col, "N/A"))
                cells.append(html.Td(value, style={"padding": index_styles["cell"]["padding'

            index_rows.append(html.Tr(cells))

        # Create the indicator table
        indicator_table = html.Table(
            [indicator_header] + index_rows,
            style=index_styles["table"]
        )

        # Return both tables in a container
        return html.Div([forecast_table, indicator_table])
```

7.3 Modals.py

```
"""
Modal Dialog Callbacks Module

This module contains all callbacks related to modal dialogs:
- Metric selection modal
- Scenario selection modal
- Extract data modal
- Content and sidebar visibility management
"""

import dash
from dash import Input, Output, State, html, dcc
from dash.exceptions import PreventUpdate

# Import configuration and data
from ..config_and_data import (
    config,
    df_index,
    institute_view_text,
    frv_text,
    get_dashboard_file_path
)

from ..layout import (
    actual_content,
    historical_content,
    about_content
)
```

```
import pandas as pd
# Import utility functions
from ..utils import create_score_display


def register_modal_callbacks(app, cache):
    """
    Register all modal-related callbacks with the app.

    Args:
        app: The Dash application instance
        cache: The Flask-Cache instance
    """

    # Callback to toggle visibility of selection row based on active view
    @app.callback(
        Output("selection-row", "style"),
        [Input("btn-actual", "n_clicks"),
         Input("btn-historical", "n_clicks"),
         Input("btn-about", "n_clicks")]
    )
    def toggle_selection_row(actual_clicks, historical_clicks, about_clicks):
        """
        Hide/show the selection row (metric, scenario, extract) based on active view.
        """
        ctx = dash.callback_context

        # If no trigger (first execution), assume we're in Actual view
        if not ctx.triggered:
            return {}

        triggered_id = ctx.triggered[0]["prop_id"].split(".")[0]

        # Show row only in Actual view
        if triggered_id == "btn-actual":
            return {}
        else:
            return {"display": "none"}

    # Callback to update content on tab change and handle sidebar visibility
    @app.callback(
        [Output("content", "children"),
         Output("country-weight", "children"),
         Output("country-sectors", "children"),
         Output("country-scores", "children"),
         Output("right-sidebar", "style")],
```

```
    [Input("btn-actual", "n_clicks"),
     Input("btn-historical", "n_clicks"),
     Input("btn-about", "n_clicks"),
     Input("dropdown-country", "value"),
     Input("scenario-info-btn", "n_clicks")]
)
def update_content(btn_actual, btn_historical, btn_about, selected_country, scenario_inf
    """
    Update the main content area based on the selected navigation tab.
    Hides the sidebar when on About or Historical tabs.

    Returns:
        tuple: (content, country_weight, country_sectors, country_scores, sidebar_style,
    """
    ctx = dash.callback_context
    if not ctx.triggered:
        return actual_content, "", "", "", {}

    triggered_id = ctx.triggered[0]["prop_id"].split(".")[0]

    # For About, Historical or Info tabs, hide the sidebar
    if triggered_id in ["btn-about", "btn-historical", "scenario-info-btn"]:
        if triggered_id in ["btn-about", "scenario-info-btn"]:
            # When About button is pressed, load content directly here
            about_with_md = about_content
            # Update markdown content directly
            for child in about_with_md.children:
                if hasattr(child, 'id') and child.id == 'institute-view-content':
                    child.children = institute_view_text
            return about_with_md, "", "", "", {"display": "none"}
        elif triggered_id == "btn-historical":
            # When Historical button is pressed, load content directly here
            hist_with_md = historical_content
            # Update markdown content directly
            for child in hist_with_md.children:
                if hasattr(child, 'id') and child.id == 'frv-content':
                    child.children = frv_text
            return hist_with_md, "", "", "", {"display": "none"}

    # Handle Actual tab or country dropdown change
    if triggered_id in ["btn-actual", "dropdown-country"]:
        if selected_country:
            if selected_country == "All":
                # For "All", don't show weight, stocks and scores
                country_weight = ""
                country_sectors = ""
```

```
                country_scores = ""
            else:
                # Get weight and stocks from index data
                row_idx = df_index[df_index["Country"] == selected_country]
                if not row_idx.empty:
                    weight_value = row_idx.iloc[0].get("Weight")
                    if weight_value is not None:
                        weight_value = weight_value * 100
                    stocks_value = row_idx.iloc[0].get("Number of stocks")
                    country_weight = f"Weight of {selected_country}: {weight_value:.1f}%
                    country_sectors = f"Number of stocks: {stocks_value:.0f}" if stocks_
                else:
                    country_weight = "Weight of the Country: N/A"
                    country_sectors = "Number of stocks: N/A"

                # Read dashboard for scores
                try:
                    dashboard_file = get_dashboard_file_path()
                    df_dash = pd.read_excel(dashboard_file, sheet_name="Dashboard")
                    df_country_dash = df_dash[df_dash["Country"] == selected_country]
                    if not df_country_dash.empty:
                        row_dash = df_country_dash.iloc[0]
                        score_total = row_dash.get("TotalScore")
                        score_val = row_dash.get("Valuation Score")
                        score_risk = row_dash.get("Risk Score")
                        score_fund = row_dash.get("Fundamental Score")
                    else:
                        score_total = score_val = score_risk = score_fund = None
                except Exception as e:
                    score_total = score_val = score_risk = score_fund = None

                # Create score components with color coding
                p_total = create_score_display("Total Score", score_total)
                p_val = create_score_display("Valuation Score", score_val)
                p_risk = create_score_display("Risk Score", score_risk)
                p_fund = create_score_display("Fundamental Score", score_fund)
                country_scores = html.Div([p_total, p_val, p_risk, p_fund])
        else:
            country_weight = ""
            country_sectors = ""
            country_scores = ""
        return actual_content, country_weight, country_sectors, country_scores, {}

    # Default return
    return actual_content, "", "", "", {}
```

```python
# Callback to toggle visibility of charts based on country selection
@app.callback(
    [Output("polar-plot-col", "style"),
     Output("bar-plot-col", "style")],
    [Input("dropdown-country", "value")]
)
def hide_plots_if_all(selected_country):
    """
    Hide the polar and bar plots when 'All' is selected.
    """
    if not selected_country or selected_country == "All":
        return ({"display": "none"}, {"display": "none"})
    else:
        return ({}, {})

# Callback to update accordion visibility in the sidebar
@app.callback(
    [Output("country-data-accordion", "style"),
     Output("economic-data-accordion", "style"),
     Output("forecast-accordion", "style")],
    [Input("dropdown-country", "value")]
)
def update_accordion_visibility(selected_country):
    """
    Hide country data accordions when 'All' is selected.
    """
    if selected_country == "All":
        return [{"display": "none"}, {"display": "none"}, {"display": "none"}]
    else:
        return [{}, {}, {}]

# Callback for Metric selection modal
@app.callback(
    [Output("selected-metric", "data"),
     Output("metric-modal", "is_open")],
    [Input("metric-link", "n_clicks"),
     Input("close-metric-modal", "n_clicks"),
     Input("metric-option-all", "n_clicks"),
     Input("metric-option-fundamentals", "n_clicks"),
     Input("metric-option-risks", "n_clicks"),
     Input("metric-option-valuations", "n_clicks"),
     Input("dropdown-country", "value"),
     Input("btn-about", "n_clicks"),
     Input("btn-historical", "n_clicks")],
    [State("selected-metric", "data"),
     State("metric-modal", "is_open")]
```

```
)
def update_metric_modal(n_link, n_close, n_all, n_fund, n_risks, n_val,
                        country_value, about_clicks, historical_clicks,
                        current_metric, is_open):
    """
    Handle the metric selection modal interactions.
    """
    ctx = dash.callback_context

    # If no trigger (first execution), don't update
    if not ctx.triggered:
        raise PreventUpdate

    triggered_id = ctx.triggered[0]['prop_id'].split('.')[0]

    # Close modal when switching views
    if triggered_id in ["btn-about", "btn-historical"]:
        return current_metric, False

    # When country changes, reset to "All" but keep modal closed
    if triggered_id == "dropdown-country":
        return "All", False

    # Only explicit click on metric button should open modal
    if triggered_id == "metric-link" and n_link is not None and n_link > 0:
        return current_metric, True

    # Close modal and update value when option selected
    if triggered_id == "close-metric-modal":
        return current_metric, False
    if triggered_id == "metric-option-all":
        return "All", False
    if triggered_id == "metric-option-fundamentals":
        return "Fundamentals", False
    if triggered_id == "metric-option-risks":
        return "Risks", False
    if triggered_id == "metric-option-valuations":
        return "Valuations", False

    # For all other cases, don't update
    raise PreventUpdate

# Callback for Scenario selection modal
@app.callback(
    [Output("selected-scenario", "data"),
     Output("scenario-modal", "is_open")],
```

```
    [Input("scenario-link", "n_clicks"),
     Input("close-scenario-modal", "n_clicks")] +
    [Input(f"scenario-option-{scenario['id']}", "n_clicks") for scenario in config["scen
    [Input("btn-about", "n_clicks"),
     Input("btn-historical", "n_clicks"),
     Input("dropdown-country", "value")],
    [State("selected-scenario", "data"),
     State("scenario-modal", "is_open")]
)
def update_scenario_modal(*args):
    """
    Handle the scenario selection modal interactions.
    """
    # Extract arguments
    num_options = len(config["scenarios"]["list"])
    n_link, n_close = args[0:2]
    option_clicks = args[2:2+num_options]
    about_clicks, historical_clicks, country_value = args[2+num_options:5+num_options]
    current_scenario, is_open = args[-2:]

    ctx = dash.callback_context

    # If no trigger (first execution), don't update
    if not ctx.triggered:
        raise PreventUpdate

    triggered_id = ctx.triggered[0]['prop_id'].split('.')[0]

    # Close modal when switching views or changing country
    if triggered_id in ["btn-about", "btn-historical", "dropdown-country"]:
        return current_scenario, False

    # Only explicit click on scenario button should open modal
    if triggered_id == "scenario-link" and n_link is not None and n_link > 0:
        return current_scenario, True

    # Close modal when Close button clicked
    if triggered_id == "close-scenario-modal":
        return current_scenario, False

    # Handle scenario options
    scenario_ids = [scenario["id"] for scenario in config["scenarios"]["list"]]
    for i, scenario_id in enumerate(scenario_ids):
        if triggered_id == f"scenario-option-{scenario_id}" and option_clicks[i] is not
            return scenario_id, False
```

```
    # For all other triggers, don't update
    raise PreventUpdate

# Callback to update the Metric link button text
@app.callback(
    Output("metric-link", "children"),
    [Input("selected-metric", "data")]
)
def update_metric_link(selected_metric):
    """Update the metric link button text to show the selected metric."""
    return selected_metric

# Callback to update the Scenario link button text
@app.callback(
    Output("scenario-link", "children"),
    [Input("selected-scenario", "data")]
)
def update_scenario_link(selected_scenario):
    """Update the scenario link button text to show the selected scenario."""
    # Find the scenario name from the ID
    for scenario in config["scenarios"]["list"]:
        if scenario["id"] == selected_scenario:
            return scenario["name"]

    # Default fallback
    return "Standard" if not selected_scenario else selected_scenario.capitalize()

# Callback to toggle the Extract Data modal
@app.callback(
    Output("extract-modal", "is_open"),
    [Input("extract-data-button", "n_clicks"),
     Input("close-extract-modal", "n_clicks"),
     Input("btn-about", "n_clicks"),
     Input("btn-historical", "n_clicks")],
    [State("extract-modal", "is_open")]
)
def toggle_extract_modal(n_open, n_close, about_clicks, historical_clicks, is_open):
    """Toggle the extract data modal visibility."""
    ctx = dash.callback_context
    if not ctx.triggered:
        raise PreventUpdate

    triggered_id = ctx.triggered[0]['prop_id'].split('.')[0]

    # Close modal when switching views
    if triggered_id in ["btn-about", "btn-historical"]:
```

```
            return False

        # Open/close modal only when needed
        if triggered_id == "extract-data-button" and n_open:
            return True
        elif triggered_id == "close-extract-modal" and n_close:
            return False

        # In all other cases, don't modify state
        raise PreventUpdate

    # Callback to update the country flag image
    @app.callback(
        Output("country-flag", "src"),
        [Input("dropdown-country", "value")]
    )
    def update_country_flag(selected_country):
        """Update the country flag image based on selected country."""
        if not selected_country or selected_country == "All":
            return ""
        mapping = config["countries"]["mapping"].get(selected_country, {})
        return mapping.get("flag_url", "")
```

7.4 Plots.py

```
"""
Ristrutturazione del file callback_plots.py per rendere le funzioni accessibili a livello d
"""

"""
Plot and Visualization Callbacks Module

This module contains all callbacks related to generating and updating plots:
- Polar plot
- Bar plot
- Financial charts
- Multiple box plot
- Stacked bar charts for total return decomposition
"""

import dash
from dash import Input, Output, dcc, html
import pandas as pd
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import math
```

```python
import os
from datetime import datetime
import dash_table

# Import configuration and utilities
from ..config_and_data import (
    config,
    df_variables,
    metric_colors,
    metric_labels,
    get_dashboard_file_path,
    valid_fund,
    valid_risk,
    valid_val
)

from ..utils import lighten_color, order_dataframe_by_country
from ..layout import make_stacked_bar_for_all_indices, make_stacked_bar_for_country, make_to

# Definire le funzioni a livello di modulo anziché all'interno di register_plot_callbacks

def create_polar_plot(selected_country, selected_metric):
    """
    Generate polar plot visualization based on selected country and metric.

    Returns:
        dict: Plotly figure object
    """
    if not selected_country or selected_country == "All":
        return go.Figure(layout={'annotations': [{
            'text': config["messages"]["errors"]["no_country_selected"],
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    dashboard_file = get_dashboard_file_path()

    if not os.path.exists(dashboard_file):
        return go.Figure(layout={'annotations': [{
            'text': config["messages"]["errors"]["dashboard_file_not_found"],
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    df_dash = pd.read_excel(dashboard_file, sheet_name="Dashboard")
    df_country = df_dash[df_dash["Country"] == selected_country]
```

```
if df_country.empty:
    return go.Figure(layout={'annotations': [{
        'text': config["messages"]["errors"]["no_data_for_country"],
        'xref': 'paper', 'yref': 'paper',
        'showarrow': False, 'font': {'size': 16}
    }]})

# Exclude certain columns and melt the dataframe for plotting
exclude_cols = config["data_processing"]["excluded_columns"]
df_plot = df_country.drop(columns=exclude_cols, errors="ignore")
df_melt = df_plot.melt(var_name="Variable", value_name="Value")

if selected_metric and selected_metric != "All":
    valid_vars = df_variables[df_variables["F/R/V"] == selected_metric]["What"].unique()
    df_melt = df_melt[df_melt["Variable"].isin(valid_vars)]

if df_melt.empty:
    return go.Figure(layout={'annotations': [{
        'text': config["messages"]["errors"]["no_valid_data"],
        'xref': 'paper', 'yref': 'paper',
        'showarrow': False, 'font': {'size': 16}
    }]})

df_merged = df_melt.merge(
    df_variables[["What", "F/R/V"]],
    left_on="Variable",
    right_on="What",
    how="left"
)

# Get polar chart settings
polar_config = config["charts"]["layout"]["polar"]

unique_vars = df_merged["Variable"].unique().tolist()
n_vars = len(unique_vars)
angle_map = {var: i * (360 / n_vars) for i, var in enumerate(unique_vars)}
bar_width = polar_config["bar_width_factor"] * (360 / n_vars)

fig = go.Figure()
for cat in df_merged["F/R/V"].dropna().unique():
    sub = df_merged[df_merged["F/R/V"] == cat]
    if sub.empty:
        continue
    r_values = []
    theta_values = []
```

```
        hover_text = []
        for _, row in sub.iterrows():
            r_values.append(row["Value"])
            theta_values.append(angle_map[row["Variable"]])
            hover_text.append(f"{row['Variable']}: {row['Value']}")
        fig.add_trace(go.Barpolar(
            r=r_values,
            theta=theta_values,
            width=[bar_width] * len(r_values),
            name=metric_labels.get(cat, cat),
            marker_color=metric_colors.get(cat, "gray"),
            marker_line_color=polar_config["marker_line"]["color"],
            marker_line_width=polar_config["marker_line"]["width"],
            opacity=polar_config["opacity"],
            hovertext=hover_text,
            hovertemplate="%{hovertext}<extra></extra>"
        ))

    tickvals = [angle_map[var] for var in unique_vars]
    ticktext = unique_vars

    # Use the Polar layout settings from the config
    default_title = polar_config["default_title"].format(country=selected_country)
    common_layout = config["charts"]["layout"]["common"]

    fig.update_layout(
        title=default_title,
        showlegend=False,
        polar=dict(
            radialaxis=polar_config["radialaxis"],
            angularaxis=dict(
                showticklabels=polar_config["angularaxis"]["showticklabels"],
                ticks=polar_config["angularaxis"]["ticks"],
                tickmode='array',
                tickvals=tickvals,
                ticktext=ticktext,
                rotation=polar_config["angularaxis"]["rotation"],
                direction=polar_config["angularaxis"]["direction"]
            )
        ),
        template=None,
        plot_bgcolor=common_layout["plot_bgcolor"],
        paper_bgcolor=common_layout["paper_bgcolor"]
    )

    return fig
```

```
def create_bar_plot(selected_country, scenario):
    """
    Generate bar plot visualization comparing country scores with averages.

    Returns:
        dict: Plotly figure object
    """
    if not selected_country or selected_country == "All":
        return go.Figure(layout={'annotations': [{
            'text': config["messages"]["errors"]["no_country_selected"],
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    dashboard_file = get_dashboard_file_path()

    if not os.path.exists(dashboard_file):
        return go.Figure(layout={'annotations': [{
            'text': config["messages"]["errors"]["dashboard_file_not_found"],
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    df_dash = pd.read_excel(dashboard_file, sheet_name="Dashboard")
    df_country = df_dash[df_dash["Country"] == selected_country]

    # Handle empty dataframe case
    if df_country.empty:
        selected_values = {"Fundamental Score": 0, "Risk Score": 0, "Valuation Score": 0, "T
    else:
        row = df_country.iloc[0]
        fundamental = row.get("Fundamental Score", 0)
        risk = row.get("Risk Score", 0)
        valuation = row.get("Valuation Score", 0)

        # Apply scenario weights if not standard
        scenario_weights = config["scenarios"]["weights"]
        if scenario not in scenario_weights:
            scenario = "standard"

        weights = scenario_weights[scenario]

        if scenario == "standard":
            selected_values = {
                "Fundamental Score": fundamental,
```

```
                "Risk Score": risk,
                "Valuation Score": valuation,
                "TotalScore": row.get("TotalScore", 0)
            }
        else:
            # Calculate weighted total based on scenario
            total = (fundamental * (weights["fundamental"] / 100) +
                     risk * (weights["risk"] / 100) +
                     valuation * (weights["valuation"] / 100))
            selected_values = {
                "Fundamental Score": fundamental,
                "Risk Score": risk,
                "Valuation Score": valuation,
                "TotalScore": total
            }

    # Calculate averages
    if scenario == "standard":
        averages = {
            "Fundamental Score": df_dash["Fundamental Score"].mean(),
            "Risk Score": df_dash["Risk Score"].mean(),
            "Valuation Score": df_dash["Valuation Score"].mean(),
            "TotalScore": df_dash["TotalScore"].mean()
        }
    else:
        # Calculate weighted averages based on scenario
        weights = scenario_weights[scenario]
        averages = {
            "Fundamental Score": df_dash["Fundamental Score"].mean(),
            "Risk Score": df_dash["Risk Score"].mean(),
            "Valuation Score": df_dash["Valuation Score"].mean(),
            "TotalScore": df_dash.apply(
                lambda r: r["Fundamental Score"] * (weights["fundamental"] / 100) +
                          r["Risk Score"] * (weights["risk"] / 100) +
                          r["Valuation Score"] * (weights["valuation"] / 100), axis=1
            ).mean()
        }

    # Get bar chart configuration
    bar_config = config["charts"]["layout"]["bar"]
    bar_colors = bar_config["colors"]

    # Create the selected country bars
    trace_selected = go.Bar(
        x=list(selected_values.keys()),
        y=[selected_values[m] for m in selected_values],
```

```
            name=f"{selected_country}",
            marker_color=[bar_colors[m] for m in selected_values]
        )

        # Create the average bars with lightened colors and pattern
        trace_average = go.Bar(
            x=list(averages.keys()),
            y=[averages[m] for m in averages],
            name="Average",
            marker=dict(
                color=[lighten_color(bar_colors[m], bar_config["lighten_factor"]) for m in avera
                pattern=dict(
                    shape=bar_config["average_pattern"]["shape"]
                )
            )
        )

        # Create the figure
        fig = go.Figure(data=[trace_selected, trace_average])

        # Get common layout settings
        common_layout = config["charts"]["layout"]["common"]
        legend_position = common_layout["legend_position"]

        # Use Bar plot layout settings from config
        default_title = bar_config["default_title"].format(country=selected_country)
        fig.update_layout(
            barmode="group",
            title=default_title,
            showlegend=False,
            legend=dict(
                orientation=legend_position["orientation"],
                yanchor=legend_position["yanchor"],
                y=legend_position["y"],
                xanchor=legend_position["xanchor"],
                x=legend_position["x"]
            ),
            plot_bgcolor=common_layout["plot_bgcolor"],
            paper_bgcolor=common_layout["paper_bgcolor"]
        )

        return fig

    def create_financial_chart(selected_country):
        """
        Generate financial chart comparing country index with MXEF index.
```

```
    Returns:
        dict: Plotly figure object
    """
    if not selected_country or selected_country == "All":
        return go.Figure(layout={'annotations': [{
            'text': config["messages"]["errors"]["no_country_selected"],
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    try:
        mapping = config["countries"]["mapping"].get(selected_country, {})
        sheet_name = mapping.get("sheet_name", "")
        if not sheet_name:
            return go.Figure(layout={'annotations': [{
                'text': config["messages"]["errors"]["msci_index_not_found"],
                'xref': 'paper', 'yref': 'paper',
                'showarrow': False, 'font': {'size': 16}
            }]})

        df_country = pd.read_excel(
            config["file_paths"]["bloomberg"],
            sheet_name=sheet_name,
            skiprows=config["file_paths"]["bloomberg_skiprows"]
        )
        date_col = df_country.columns[0]
        if "PX_LAST" not in df_country.columns:
            return go.Figure(layout={'annotations': [{
                'text': config["messages"]["errors"]["missing_columns"],
                'xref': 'paper', 'yref': 'paper',
                'showarrow': False, 'font': {'size': 16}
            }]})

        df_country = df_country[[date_col, "PX_LAST"]].dropna()
        df_country.columns = ["Date", "CountryPX"]

        df_mxef = pd.read_excel(
            config["file_paths"]["bloomberg"],
            sheet_name=config["charts"]["financial"]["default_indices"]["mxef_index"],
            skiprows=config["file_paths"]["bloomberg_skiprows"]
        )
        if "PX_LAST" not in df_mxef.columns:
            return go.Figure(layout={'annotations': [{
                'text': config["messages"]["errors"]["missing_mxef_index"],
                'xref': 'paper', 'yref': 'paper',
```

```
        'showarrow': False, 'font': {'size': 16}
    }]})

df_mxef = df_mxef[[date_col, "PX_LAST"]].dropna()
df_mxef.columns = ["Date", "MXEF"]

df_merged = pd.merge(df_mxef, df_country, on="Date", how="inner")

# Base 100 normalization
base_mxef = df_merged["MXEF"].iloc[0]
if base_mxef != 0:
    df_merged["MXEF"] = df_merged["MXEF"] / base_mxef * 100

base_country = df_merged["CountryPX"].iloc[0]
if base_country != 0:
    df_merged["CountryPX"] = df_merged["CountryPX"] / base_country * 100

# Create the figure
fig = go.Figure()
fig.add_trace(go.Scatter(
    x=df_merged["Date"],
    y=df_merged["MXEF"],
    mode="lines",
    name="MXEF Index",
    line=dict(color=config["app_settings"]["style"]["secondary_color"])
))
fig.add_trace(go.Scatter(
    x=df_merged["Date"],
    y=df_merged["CountryPX"],
    mode="lines",
    name=selected_country,
    line=dict(color=config["app_settings"]["style"]["primary_color"])
))

# Get common layout settings
common_layout = config["charts"]["layout"]["common"]
legend_position = common_layout["legend_position"]

# Use financial chart title from config
default_title = config["charts"]["financial"]["default_titles"]["index_comparison"]
    country=selected_country
)

fig.update_layout(
    title=default_title,
    xaxis_title="Date",
```

```
            yaxis_title="Base 100",
            autosize=True,
            paper_bgcolor=common_layout["paper_bgcolor"],
            plot_bgcolor=common_layout["plot_bgcolor"],
            legend=dict(
                orientation=legend_position["orientation"],
                yanchor=legend_position["yanchor"],
                y=legend_position["y"],
                xanchor=legend_position["xanchor"],
                x=legend_position["x"]
            )
        )

        return fig

    except Exception as e:
        return go.Figure(layout={'annotations': [{
            'text': f"{config['messages']['errors']['read_financial_data_error']} {str(e)}",
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

def create_multiple_box_plot(selected_country):
    """
    Generate multiple box plots for different metrics of the selected country.

    Returns:
        dict: Plotly figure object
    """
    if not selected_country or selected_country == "All":
        return go.Figure(layout={'annotations': [{
            'text': config["messages"]["errors"]["no_country_selected"],
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    try:
        mapping = config["countries"]["mapping"].get(selected_country, {})
        sheet_name = mapping.get("sheet_name", "")
        if not sheet_name:
            return go.Figure(layout={'annotations': [{
                'text': config["messages"]["errors"]["msci_index_not_found"],
                'xref': 'paper', 'yref': 'paper',
                'showarrow': False, 'font': {'size': 16}
            }]})
```

```
        df_bbg = pd.read_excel(
            config["file_paths"]["bloomberg"],
            sheet_name=sheet_name,
            skiprows=config["file_paths"]["bloomberg_skiprows"]
        )
    except Exception as e:
        return go.Figure(layout={'annotations': [{
            'text': f"{config['messages']['errors']['read_financial_data_error']} {str(e)}",
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    # Get columns and colors configuration for box plots
    box_plot_config = config["charts"]["box_plot"]
    multi_cols = list(box_plot_config["labels"].keys())
    colors = box_plot_config["colors"]

    # Filter columns with valid data
    valid_cols = []
    for col in multi_cols:
        if col in df_bbg.columns and not df_bbg[col].dropna().empty:
            valid_cols.append(col)

    if not valid_cols:
        return go.Figure(layout={'annotations': [{
            'text': config["messages"]["errors"]["no_valid_data"],
            'xref': 'paper', 'yref': 'paper',
            'showarrow': False, 'font': {'size': 16}
        }]})

    # Create a subplot with one row and multiple columns
    fig = make_subplots(rows=1, cols=len(valid_cols), shared_yaxes=False)

    # Get actual marker settings
    actual_marker = box_plot_config["actual_marker"]

    # Add box plots for each metric
    for i, col in enumerate(valid_cols):
        series_data = df_bbg[col].dropna()
        trace_name = box_plot_config["labels"].get(col, col)

        # Add box plot
        fig.add_trace(
            go.Box(
                x=[trace_name]*len(series_data),
                y=series_data,
```

```
                name=trace_name,
                boxmean='sd',
                boxpoints="outliers",
                whiskerwidth=0.2,
                marker_color=colors.get(col, "gray"),
                fillcolor=colors.get(col, "gray"),
                line=dict(color=colors.get(col, "gray"))
            ),
            row=1, col=i+1
        )

        # Add scatter for the latest value
        last_val = series_data.iloc[-1]
        fig.add_trace(
            go.Scatter(
                x=[trace_name],
                y=[last_val],
                name=f"Actual {col}",
                mode='markers+text',
                text=[actual_marker["text"]],
                textposition="top center",
                marker=dict(
                    color=actual_marker["color"],
                    size=actual_marker["size"],
                    symbol=actual_marker["symbol"]
                )
            ),
            row=1, col=i+1
        )

        # Hide y-axis for this subplot
        fig.update_yaxes(visible=False, row=1, col=i+1)

    # Update layout
    fig.update_layout(
        title=f"Box Plots for {selected_country}",
        template='plotly_white',
        showlegend=False,
        margin=dict(l=60, r=60, t=80, b=60)
    )

    return fig


def register_plot_callbacks(app, cache):
    """
```

```
    Register all plot-related callbacks with the app.

    Args:
        app: The Dash application instance
        cache: The Flask-Cache instance
    """

    # Callback to update the Polar Plot
    @app.callback(
        Output("polar-plot", "figure"),
        [Input("btn-actual", "n_clicks"),
         Input("dropdown-country", "value"),
         Input("selected-metric", "data")]
    )
    @cache.memoize(timeout=config["app_settings"]["cache"]["thresholds"]["plots"])
    def update_polar_plot(btn_actual, selected_country, selected_metric):
        """Wrapper around create_polar_plot that adds caching"""
        return create_polar_plot(selected_country, selected_metric)

    # Callback to update the Bar Plot
    @app.callback(
        Output("bar-plot", "figure"),
        [Input("btn-actual", "n_clicks"),
         Input("dropdown-country", "value"),
         Input("selected-scenario", "data")]
    )
    @cache.memoize(timeout=config["app_settings"]["cache"]["thresholds"]["plots"])
    def update_bar_plot(btn_actual, selected_country, scenario):
        """Wrapper around create_bar_plot that adds caching"""
        return create_bar_plot(selected_country, scenario)

    # Callback to update the Financial Chart
    @app.callback(
        Output("financial-chart", "figure"),
        [Input("dropdown-country", "value")]
    )
    @cache.memoize(timeout=config["app_settings"]["cache"]["thresholds"]["plots"])
    def update_financial_chart(selected_country):
        """Wrapper around create_financial_chart that adds caching"""
        return create_financial_chart(selected_country)

    # Callback to update the Multiple Box Plot
    @app.callback(
        Output("multiple-box-plot", "figure"),
        [Input("dropdown-country", "value")]
    )
```

```python
@cache.memoize(timeout=config["app_settings"]["cache"]["thresholds"]["plots"])
def update_multiple_box_plot(selected_country):
    """Wrapper around create_multiple_box_plot that adds caching"""
    return create_multiple_box_plot(selected_country)

# Callback to update the Financial Data container
@app.callback(
    Output("financial-data-container", "children"),
    [Input("dropdown-country", "value")]
)
def update_financial_data_container(selected_country):
    """
    Update the financial data container with appropriate charts.
    For "All", show regional indices. For specific country, show financial chart and bo

    Returns:
        list: List of Dash components
    """
    try:
        # If "All" is selected, show regional charts
        if not selected_country or selected_country == "All":
            index_list = config["charts"]["financial"]["indices"]
            figures = []

            for item in index_list:
                region = item["name"]
                sheet_name = item["sheet_name"]
                color = item["color"]

                # Load region data
                df_region = pd.read_excel(
                    config["file_paths"]["bloomberg"],
                    sheet_name=sheet_name,
                    skiprows=config["file_paths"]["bloomberg_skiprows"]
                )
                date_col = df_region.columns[0]
                df_region = df_region[[date_col, "PX_LAST"]].dropna()
                df_region.columns = ["Date", region]

                # Load MXEF data
                mxef_sheet = config["charts"]["financial"]["default_indices"]["mxef_inde
                df_mxef = pd.read_excel(
                    config["file_paths"]["bloomberg"],
                    sheet_name=mxef_sheet,
                    skiprows=config["file_paths"]["bloomberg_skiprows"]
                )
```

```
df_mxef = df_mxef[[date_col, "PX_LAST"]].dropna()
df_mxef.columns = ["Date", "MXEF Index"]

# Merge and normalize to base 100
df_merged = pd.merge(df_mxef, df_region, on="Date", how="inner")
base_mxef = df_merged["MXEF Index"].iloc[0]
df_merged["MXEF Index"] = df_merged["MXEF Index"] / base_mxef * 100 if b

base_region = df_merged[region].iloc[0]
df_merged[region] = df_merged[region] / base_region * 100 if base_regio

# Get common layout settings
common_layout = config["charts"]["layout"]["common"]
legend_position = common_layout["legend_position"]

# Create figure
fig = go.Figure()
fig.add_trace(go.Scatter(
    x=df_merged["Date"],
    y=df_merged["MXEF Index"],
    mode="lines",
    name="MXEF Index",
    line=dict(color=config["app_settings"]["style"]["secondary_color"])
))
fig.add_trace(go.Scatter(
    x=df_merged["Date"],
    y=df_merged[region],
    mode="lines",
    name=region,
    line=dict(color=color)
))
fig.update_layout(
    title=region,
    xaxis_title="Date",
    yaxis_title="Base 100",
    autosize=True,
    paper_bgcolor=common_layout["paper_bgcolor"],
    plot_bgcolor=common_layout["plot_bgcolor"],
    legend=dict(
        orientation=legend_position["orientation"],
        yanchor=legend_position["yanchor"],
        y=legend_position["y"],
        xanchor=legend_position["xanchor"],
        x=legend_position["x"]
    )
)
```

```
                figures.append(dcc.Graph(figure=fig))

            return figures
        else:
            # For specific country, show financial chart and box plot
            return [
                dcc.Graph(id="financial-chart"),
                dcc.Graph(id="multiple-box-plot")
            ]
    except Exception as e:
        return html.Div(f"Error loading Bloomberg data: {str(e)}")

# Callback to update the Total Return Decomposition section
@app.callback(
    Output("total-return-decomposition", "children"),
    [Input("dropdown-country", "value")]
)
def update_total_return_decomposition(selected_country):
    """
    Update the total return decomposition section.
    For "All", show stacked bar chart for global indices.
    For specific country, show return decomposition table and stacked bar chart.

    Returns:
        dash.html.Div: Dash component with return decomposition
    """
    if not selected_country or selected_country == "All":
        return make_stacked_bar_for_all_indices()
    else:
        return html.Div([
            make_total_return_for_country(selected_country),
            make_stacked_bar_for_country(selected_country)
        ])

# Callback to update the Flow Analysis section
@app.callback(
    Output("flow-analysis-section", "children"),
    [Input("dropdown-country", "value")]
)
def update_flow_analysis_section(selected_country):
    """
    Update the flow analysis section visibility and content.
    Only shown when "All" is selected.

    Returns:
        dash.html.Div: Dash component with flow analysis
```

```
    """
    if selected_country == "All":
        header = html.H5("Flow Analysis", className="text-left")
        # The callback for "flow-analysis-table" will handle loading the table
        table = html.Div(id="flow-analysis-table")
        return html.Div([header, table])
    else:
        return ""

# Callback to update the Flow Analysis table
@app.callback(
    Output("flow-analysis-table", "children"),
    [Input("dropdown-country", "value")]
)
def update_flow_analysis_table(selected_country):
    """
    Update the flow analysis table with Bloomberg flow data.
    Only shown when "All" is selected.

    Returns:
        dash.html.Div: Table component with flow data
    """
    if selected_country != "All":
        return ""

    try:
        df_flow = pd.read_excel(
            config["file_paths"]["bloomberg"],
            sheet_name="Flows",
            skiprows=config["file_paths"]["bloomberg_skiprows"]
        )
        columns = [{"name": str(col), "id": str(col)} for col in df_flow.columns]
        table = dash_table.DataTable(
            data=df_flow.to_dict("records"),
            columns=columns,
            page_size=10,
            style_table={'overflowX': 'auto'},
        )
        return table
    except Exception as e:
        return html.Div(f"Error reading Bloomberg flow data: {str(e)}")
```

7.5 Ppt.py

```
"""
PowerPoint Export Callbacks Module

This module contains callbacks for generating and downloading PowerPoint presentations
with visualizations and data for selected countries.
"""

import os
import io
import pandas as pd
import plotly.io as pio
from datetime import datetime
from dash import Input, Output, State, dcc
from dash.exceptions import PreventUpdate

# PowerPoint libraries
from pptx import Presentation
from pptx.util import Inches, Pt, Cm
from pptx.dml.color import RGBColor
from pptx.enum.text import PP_ALIGN

# Import configuration
from ..config_and_data import (
    config,
    df_variables,
    df_names,
    df_index,
    get_dashboard_file_path
)

from ..utils import order_dataframe_by_country


def register_ppt_callbacks(app, cache):
    """
    Register PowerPoint export-related callbacks with the app.

    Args:
        app: The Dash application instance
        cache: The Flask-Cache instance
    """

    @app.callback(
        Output("download-ppt", "data"),
```

```
    [Input("download-ppt-button", "n_clicks")],
    [State("extract-country-checklist", "value"),
     State("selected-metric", "data"),
     State("selected-scenario", "data")]
)
def generate_ppt(n_clicks, selected_countries, metric, scenario):
    """
    Generate a PowerPoint presentation with visualizations for the selected countries.

    Args:
        n_clicks: Number of times the download button was clicked
        selected_countries: List of selected countries
        metric: Selected metric filter
        scenario: Selected scenario

    Returns:
        dict: Download data for the PowerPoint file
    """
    # Don't generate if download button not clicked
    if not n_clicks:
        raise PreventUpdate

    # If "All" is selected, use all available countries
    if selected_countries and "All" in selected_countries:
        selected_countries = sorted(df_names["Country"].dropna().unique().tolist())

    # Don't generate if no countries selected
    if not selected_countries:
        raise PreventUpdate

    # Import necessary callbacks for plot generation
    from .callback_plots import (
        create_polar_plot,
        create_bar_plot,
        create_financial_chart,
        create_multiple_box_plot)
    # Get dashboard path
    dashboard_file = get_dashboard_file_path()

    # Get PowerPoint configuration
    ppt_cfg = config["ppt"]["config"]

    # Create a presentation using the theme file specified in config
    ppt_theme = config["file_paths"]["ppt_theme"]
    prs = Presentation(ppt_theme)
```

```
# --- COVER SLIDE ---
cover_layout = prs.slide_layouts[ppt_cfg["cover_layout_index"]]
cover_slide = prs.slides.add_slide(cover_layout)

# Set the cover title
if cover_slide.shapes.title:
    cover_slide.shapes.title.text = ppt_cfg["cover_title"]

# Format date for cover slide
formatted_date = datetime.now().strftime(ppt_cfg["ppt_date_format"])

# Set the cover date
if len(cover_slide.placeholders) > 1:
    cover_slide.placeholders[1].text = f"Date: {formatted_date}"

# --- TABLE SLIDES: Split the FRV table into 3 parts ---
# Load dashboard data
df_dash = pd.read_excel(dashboard_file, sheet_name="Dashboard")

if len(selected_countries) > 0:
    ordered_df = order_dataframe_by_country(df_dash, selected_countries[0])
else:
    ordered_df = df_dash

# Get column configurations for each metric type
tables_cfg = ppt_cfg["tables_config"]
fundamentals_cols = tables_cfg["fundamentals"]
valuations_cols = tables_cfg["valuations"]
risks_cols = tables_cfg["risks"]

# Define valid column sets for each metric type
from ..config_and_data import valid_fund, valid_risk, valid_val

def build_metric_table(df, base_cols, valid_set):
    """Build a table with base columns and additional columns from a valid set."""
    cols = list(base_cols)
    if "TotalScore" not in cols:
        cols.append("TotalScore")
    additional = [col for col in df.columns if col in valid_set]
    for col in additional:
        if col not in cols:
            cols.append(col)
    return df[cols].copy()

# Build tables for each metric type
df_fund = build_metric_table(ordered_df, fundamentals_cols, valid_fund)
```

```
df_val = build_metric_table(ordered_df, valuations_cols, valid_val)
df_risk = build_metric_table(ordered_df, risks_cols, valid_risk)

# Sort tables by their respective scores
if "Fundamental Score" in df_fund.columns:
    df_fund = df_fund.sort_values(by="Fundamental Score", ascending=False)
if "Valuation Score" in df_val.columns:
    df_val = df_val.sort_values(by="Valuation Score", ascending=False)
if "Risk Score" in df_risk.columns:
    df_risk = df_risk.sort_values(by="Risk Score", ascending=False)

# Add slides for the three tables
add_table_slide("Fundamentals Table", df_fund, prs, ppt_cfg)
add_table_slide("Valuations Table", df_val, prs, ppt_cfg)
add_table_slide("Risks Table", df_risk, prs, ppt_cfg)

# --- OVERVIEW SLIDE: "Total Shareholder Return" / "Regional Overview" ---
overview_layout = prs.slide_layouts[ppt_cfg["graph_layout_index"]]
overview_slide = prs.slides.add_slide(overview_layout)

# Set overview titles
overview_title = ppt_cfg["overview"]["title"]
overview_sub = ppt_cfg["overview"]["subtitle"]

if overview_slide.shapes.title:
    overview_slide.shapes.title.text = overview_title

if len(overview_slide.placeholders) > 1:
    overview_slide.placeholders[13].text = overview_sub

# Import stacked bar chart function
from ..layout import make_stacked_bar_for_all_indices

# Generate and add stacked bar chart
stacked_graph = make_stacked_bar_for_all_indices()
stacked_img_bytes = pio.to_image(stacked_graph.figure, format="png")

# Add the chart to the slide
placeholders_ov = {ph.placeholder_format.idx: ph for ph in overview_slide.placeholde
if 15 in placeholders_ov:
    ph = placeholders_ov[15]
    overview_slide.shapes.add_picture(
        io.BytesIO(stacked_img_bytes),
        ph.left, ph.top,
        width=ph.width,
        height=ph.height
```

```
        )
        # Remove the used placeholder
        try:
            ph.element.getparent().remove(ph.element)
        except Exception as e:
            print(f"Error removing overview placeholder: {e}")

    # --- SLIDES FOR EACH COUNTRY ---
    for country in selected_countries:
        # --- GRAPH SLIDE: Polar and Bar charts ---
        graph_layout = prs.slide_layouts[ppt_cfg["graph_layout_index"]]
        graph_slide = prs.slides.add_slide(graph_layout)

        # Set slide title and subtitle
        if graph_slide.shapes.title:
            graph_slide.shapes.title.text = f"{country}"

        if len(graph_slide.placeholders) > 1:
            graph_slide.placeholders[13].text = ppt_cfg["graph_slide_subtitle"]

        # Generate polar and bar plots
        polar_fig = create_polar_plot(country, metric)
        bar_fig = create_bar_plot(country, scenario)

        # Convert plots to images
        polar_img_bytes = pio.to_image(polar_fig, format="png")
        bar_img_bytes = pio.to_image(bar_fig, format="png")

        # Add plots to the slide
        placeholders_graph = {ph.placeholder_format.idx: ph for ph in graph_slide.placeh

        # Add polar plot
        if 15 in placeholders_graph:
            ph = placeholders_graph[15]
            graph_slide.shapes.add_picture(
                io.BytesIO(polar_img_bytes),
                ph.left, ph.top,
                width=ph.width,
                height=ph.height
            )
            try:
                ph.element.getparent().remove(ph.element)
            except Exception as e:
                print(f"Error removing graph placeholder 15: {e}")

        # Add bar plot
```

```
if 16 in placeholders_graph:
    ph = placeholders_graph[16]
    graph_slide.shapes.add_picture(
        io.BytesIO(bar_img_bytes),
        ph.left, ph.top,
        width=ph.width,
        height=ph.height
    )
    try:
        ph.element.getparent().remove(ph.element)
    except Exception as e:
        print(f"Error removing graph placeholder 16: {e}")

# --- FINANCIAL SLIDE: Index comparison and box plots ---
fin_layout = prs.slide_layouts[ppt_cfg["financial_layout_index"]]
fin_slide = prs.slides.add_slide(fin_layout)

# Set slide title and subtitle
if fin_slide.shapes.title:
    fin_slide.shapes.title.text = f"{country}"

if len(fin_slide.placeholders) > 1:
    fin_slide.placeholders[13].text = ppt_cfg["financial_slide_subtitle"]

# Get placeholders for financial slide
placeholders_fin = {ph.placeholder_format.idx: ph for ph in fin_slide.placeholde

# Generate financial charts
index_comp_fig = create_financial_chart(country)
multiple_fig = create_multiple_box_plot(country)

# Convert charts to images
index_img_bytes = pio.to_image(index_comp_fig, format="png")
multiple_img_bytes = pio.to_image(multiple_fig, format="png")

# Add index comparison chart
if 16 in placeholders_fin:
    ph = placeholders_fin[16]
    fin_slide.shapes.add_picture(
        io.BytesIO(index_img_bytes),
        ph.left, ph.top,
        width=ph.width,
        height=ph.height
    )
    try:
        ph.element.getparent().remove(ph.element)
```

```
            except Exception as e:
                print(f"Error removing financial placeholder 16: {e}")

        # Add box plots
        if 17 in placeholders_fin:
            ph = placeholders_fin[17]
            fin_slide.shapes.add_picture(
                io.BytesIO(multiple_img_bytes),
                ph.left, ph.top,
                width=ph.width,
                height=ph.height
            )
            try:
                ph.element.getparent().remove(ph.element)
            except Exception as e:
                print(f"Error removing financial placeholder 17: {e}")

    # --- FOOTER AND SLIDE NUMBERING ---
    # Format the date for the footer
    footer_date = datetime.now().strftime(ppt_cfg["ppt_filename_date_format"])

    # Get footer configuration
    footer_cfg = ppt_cfg["footer"]
    footer_text = footer_cfg["text"]
    footer_font_size = footer_cfg["font_size"]
    footer_color = footer_cfg["color"]

    # Add footer and slide numbers to all non-cover slides
    non_cover_slide_number = 1
    for slide in prs.slides:
        # Skip cover slide
        if slide.slide_layout == prs.slide_layouts[ppt_cfg["cover_layout_index"]]:
            continue

        # Process placeholders
        for ph in slide.placeholders:
            # Add footer text
            if ph.placeholder_format.idx == ppt_cfg["footer_placeholder_idx"]:
                ph.text = f"{footer_text} | {footer_date}"
                if ph.text_frame:
                    for paragraph in ph.text_frame.paragraphs:
                        for run in paragraph.runs:
                            run.font.size = Pt(footer_font_size)
                            run.font.color.rgb = RGBColor(*footer_color)

            # Add slide number
```

```
                elif ph.placeholder_format.idx == ppt_cfg["slide_number_placeholder_idx"]:
                    ph.text = f"Slide {non_cover_slide_number}"

            non_cover_slide_number += 1

        # Save the presentation to a BytesIO object
        ppt_stream = io.BytesIO()
        prs.save(ppt_stream)
        ppt_stream.seek(0)

        # Return the file for download
        return dcc.send_bytes(
            ppt_stream.read(),
            filename=f"MacroExtraction_{footer_date}.pptx"
        )


def add_table_slide(slide_title, df_table, prs, ppt_cfg):
    """
    Create a slide with a data table using the style defined in the configuration.

    Args:
        slide_title: Title for the slide
        df_table: DataFrame containing the table data
        prs: PowerPoint presentation object
        ppt_cfg: PowerPoint configuration dictionary
    """
    # Get styles for header and data from configuration
    table_header_style = ppt_cfg["table_styles"]["header"]
    table_data_style = ppt_cfg["table_styles"]["data"]

    # Create the slide
    table_layout = prs.slide_layouts[ppt_cfg["table_layout_index"]]
    slide = prs.slides.add_slide(table_layout)

    # Set the slide title
    if slide.shapes.title:
        slide.shapes.title.text = slide_title

    # Set the subtitle if placeholder exists
    if len(slide.placeholders) > 1:
        slide.placeholders[13].text = ppt_cfg["table_slide_title"]

    # Define table position and dimensions
    left = Cm(1)
    top = Cm(3)
```

```
width = Cm(23.1)
height = Cm(8.98)

# Get table dimensions from DataFrame
rows, cols = df_table.shape

# Create the table
table_obj = slide.shapes.add_table(rows + 1, cols, left, top, width, height).table

# Set row heights
table_obj.rows[0].height = Cm(table_header_style["first_col_width_cm"])  # Header row he
for i in range(rows):
    table_obj.rows[i + 1].height = Cm(table_data_style["first_col_width_cm"])  # Data ro

# Format header row
for j, col in enumerate(df_table.columns):
    cell = table_obj.cell(0, j)
    cell.text = str(col)

    # Format text in the cell
    for paragraph in cell.text_frame.paragraphs:
        for run in paragraph.runs:
            run.font.size = Pt(table_header_style["first_col_font_size"])
            run.font.color.rgb = RGBColor(*table_header_style["font_color"])

        # Set text alignment
        align = table_header_style["alignment"].lower()
        if align == "center":
            paragraph.alignment = PP_ALIGN.CENTER
        elif align == "right":
            paragraph.alignment = PP_ALIGN.RIGHT
        else:
            paragraph.alignment = PP_ALIGN.LEFT

    # Set cell background color
    cell.fill.solid()
    cell.fill.fore_color.rgb = RGBColor(*table_header_style["bg_color"])

# Format data rows
for i in range(rows):
    for j in range(cols):
        cell = table_obj.cell(i + 1, j)

        # Format cell value
        val = df_table.iloc[i, j]
        try:
```

```
                numeric_value = int(float(val))
                cell_text = str(numeric_value)
            except (ValueError, TypeError):
                cell_text = str(val)

            cell.text = cell_text

            # Format text in the cell
            for paragraph in cell.text_frame.paragraphs:
                for run in paragraph.runs:
                    # Use different font size for first column vs other columns
                    if j == 0:
                        run.font.size = Pt(table_data_style["first_col_font_size"])
                    else:
                        run.font.size = Pt(table_data_style["other_cols_font_size"])

                    run.font.color.rgb = RGBColor(*table_data_style["font_color"])

                # Set text alignment
                align = table_data_style["alignment"].lower()
                if align == "center":
                    paragraph.alignment = PP_ALIGN.CENTER
                elif align == "right":
                    paragraph.alignment = PP_ALIGN.RIGHT
                else:
                    paragraph.alignment = PP_ALIGN.LEFT

            # Set cell background color
            cell.fill.solid()
            cell.fill.fore_color.rgb = RGBColor(*table_data_style["bg_color"])
```