

Description

Our application is a cloud-based tool that uses a stock market trading algorithm engine called QuantConnect LEAN to run backtests on user-provided algorithms. The application gets inputs (date range, ticker name, stock data, algorithm file (C# or Python), algorithm parameters) from the user and stores them in a database. It then creates a new QuantConnect LEAN project, or updates an existing one. The engine executes a backtest on the project using the provided stock data, after which the backtest results are saved in a database. Users can select any backtest result using its name and display the results. The results data is shown in tables and graphs which the user can use to understand whether their algorithm is profitable or not.

Motivations

The application is used to save time to rewrite an algorithm with the intention to change only the values of its parameters. It also calculates data from any saved backtest result and display to users information which is not provided on the QuantConnect website. Its interface is simple and user-friendly.

As a programmer, our group can use this application to show hiring employers what we have done to gain the experience needed for a software engineer. It is a product that has been created from scratch to finish within 2 months. It is a solution to a real life problem.

Features

The basic features of our application include:

- Creating a new QuantConnect LEAN project by uploading an algorithm file to the application
- Running a backtest on a project with custom start/end dates, stock market data, and algorithm parameters using the QuantConnect LEAN CLI
- Storing backtest results produced by the QuantConnect LEAN CLI with relation to a project
- Display backtest results to the user on-demand with custom plots

Images and Captions

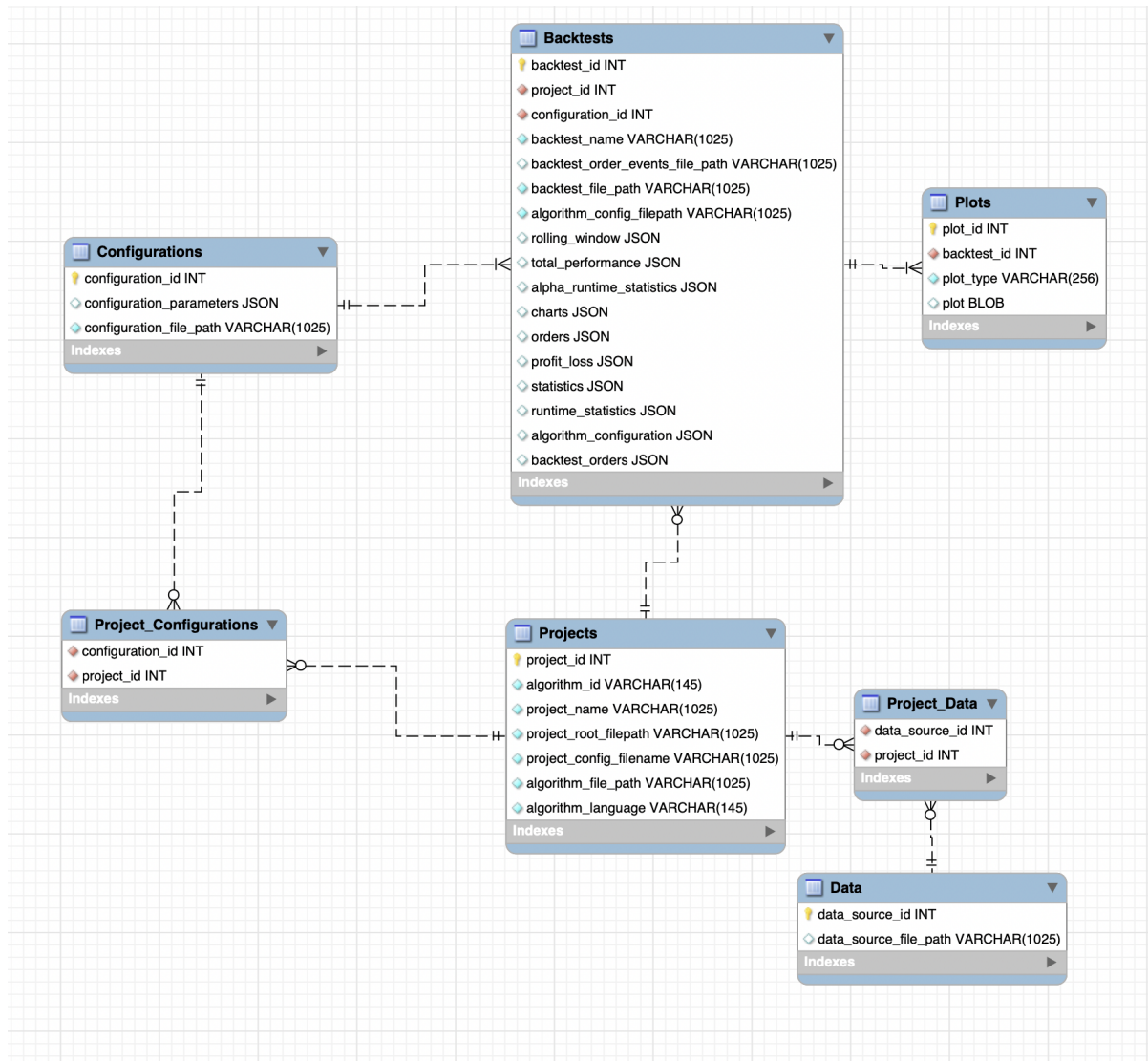
<<<BELOW>>>

```

yourapplication.classes.Backtests.Backtest
__init__(self, **kwargs)
run_backtest(self, project_root_filepath)
write_configurations_to_config(self)
insert_backtest_query(self)
select_backtest_from_db(self)
insert_configuration_query(self)
insert_project_configurations_query(self)
convert_attributes_to_dictionary_from_db(self, backtest_pull_data)
convert_backtest_to_string_for_db(self)
separate_date_to_day_month_year(self, date)
set_backtest_id(self, backtest_id)
set_project_id(self, project_id)
set_configuration_id(self, configuration_id)
set_backtest_name(self, backtest_name)
set_backtest_order_events_filename(self,
set_backtest_filename(self, backtest_filename)
set_algorithm_config_filepath(self, config_filepath)
set_rolling_window(self, rolling_window_json_object)
set_total_performance(self, total_performance_json_object)
set_alpha_runtime_statistics(self, alpha_runtime_statistics_json_object)
set_charts(self, charts_json_object)
set_orders(self, orders_json_object)
set_profit_loss(self, profit_loss_json_object)
set_statistics(self, statistics_json_object)
set_runtime_statistics(self, runtime_statistics_json_object)
set_algorithm_configuration(self, algorithm_configuration_json_object)
set_backtest_order_events(self, backtest_order_events_json_object)
set_configuration_parameters(self)
__format_profit_loss_table(self)
__format_statistics_table(self)
__format_alpha_runtime_statistics_table(self)
__format_runtime_statistics_table(self)
make_orders_table(self)
make_total_perf_table(self)
make_charts_table(self)
make_rolling_window_table(self)

```

The Backtest class is responsible for collecting the output data as a result of running a “lean backtest.” Here we can pull backtest data on a previously run project from the database and format it for display in a table, or pass it to the Plots class for building and displaying graphs. The backtest class is also responsible for inserting new backtest data into the database for later retrieval.















The backend database is the backbone of the web application. The class structure is modeled after the database. The database is responsible for storing json results, file paths, and outlines the relationships between the classes/entities that encapsulate the project.

c yourapplication.classes.Plot.Plot

- m `__init__(self, **kwargs)`
- m `create_plot(self)`
- m `insert_plot_query(self)`
- m `select_plot_ids_from_backtest_id_query(self)`
- m `select_plot_from_plot_id(self)`
- m `set_plot_id(self, plot_id)`
- m `set_plot_file(self, plot_file)`

The Plot class is responsible for building and storing graph displays of the backtest. Each Backtest can/will have multiple plots, with each plot relating to one backtest.

yourapplication.classes.Projects.Project

-  `__init__(self, **kwargs)`
-  `create_project(self, algorithm_file, language)`
-  `insert_project_query(self)`
-  `select_project_query(self)`
-  `store_query_in_member_attributes(self, project)`
-  `set_project_id(self, project_id)`
-  `set_algorithm_id(self, algorithm_id)`
-  `set_project_name(self, project_name)`
-  `set_project_root_filepath(self, project_root_filepath)`
-  `set_algorithm_file_path(self, algorithm_file_path)`
-  `set_algorithm_language(self, algorithm_language)`
-  `set_project_config_filename(self, config_filepath)`

The Projects class acts as sort of the entry point for the whole project. The user uploads a new trading algorithm and a new project is created. The Projects class tracks details about the specific project such as - both the “project” root path, project configuration file, and algorithm file paths, and its project_id after insertion into the database. The Project class creates a project that can later be accessed for running a backtest against with user defined parameters.

```
#####
#                                                                    #
#                                                                    #
#                               Configs                               #
#                                                                    #
#                                                                    #
#####

dirname = os.path.dirname(__file__)
PROJECTS_FOLDER = '/projects/'
BACKTEST_FOLDER = '/projects/backtests/'

app.config['PROJECTS_FOLDER'] = PROJECTS_FOLDER
app.config['BACKTEST_FOLDER'] = BACKTEST_FOLDER
nav = Navigation()

nav.Bar('top', [...])
__WEBAPP_DEBUG_SWITCH = True
nav.init_app(app)

#####
#                                                                    #
#                                                                    #
#                               Routes                               #
#                                                                    #
#                                                                    #
#####

@app.route('/')
def index():...

@app.route('/load-project')
def load_project():...

# Create a new project from algorithm
@app.route('/create-project', methods=['POST', 'GET'])
def create_project():...

@app.route('/project-selector', methods=['GET', 'POST'])
def run_project():...

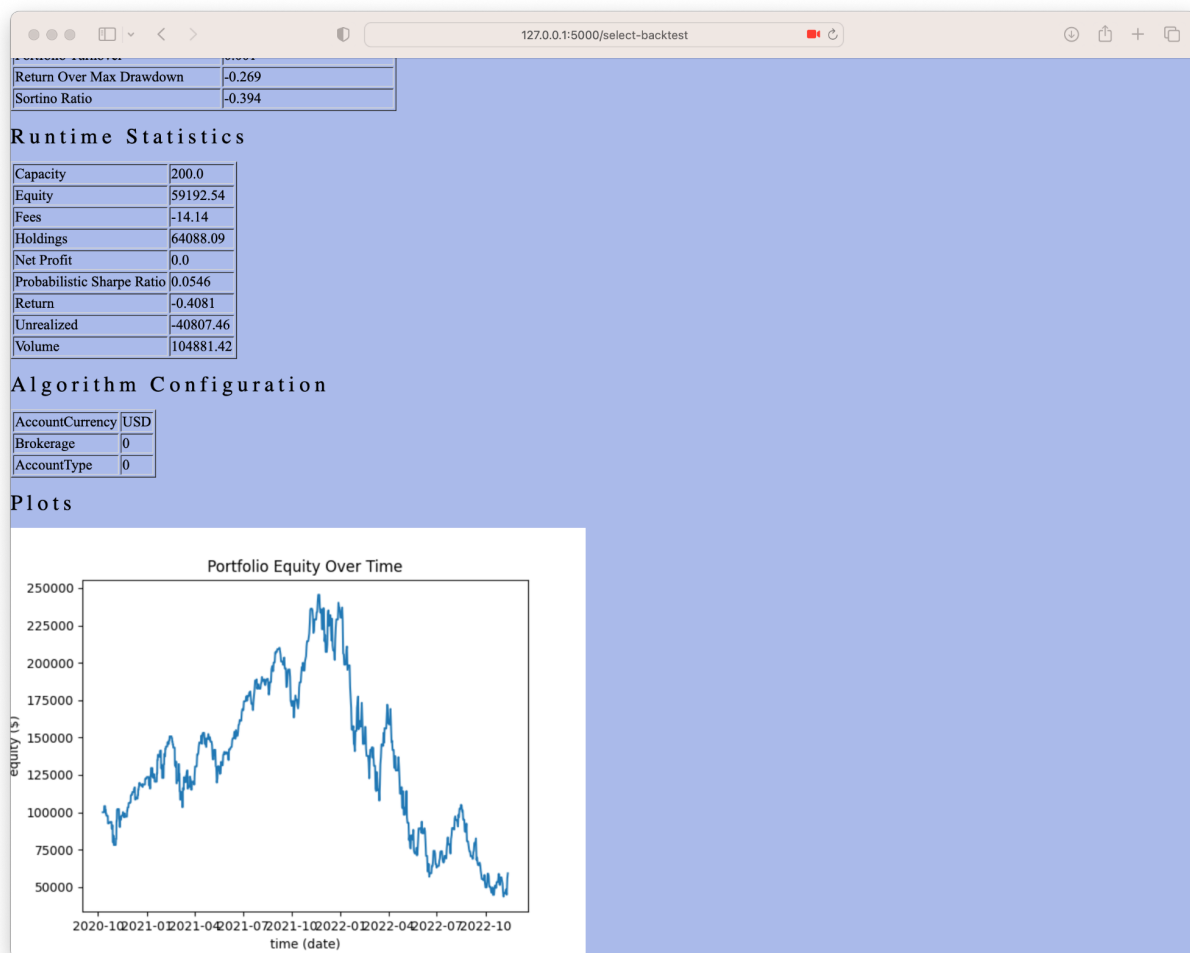
@app.route('/select-backtest', methods=['GET', 'POST'])
def select_backtest():...

@app.route('/plots/<plot_id>', methods=['GET'])
def get_plot(plot_id):...

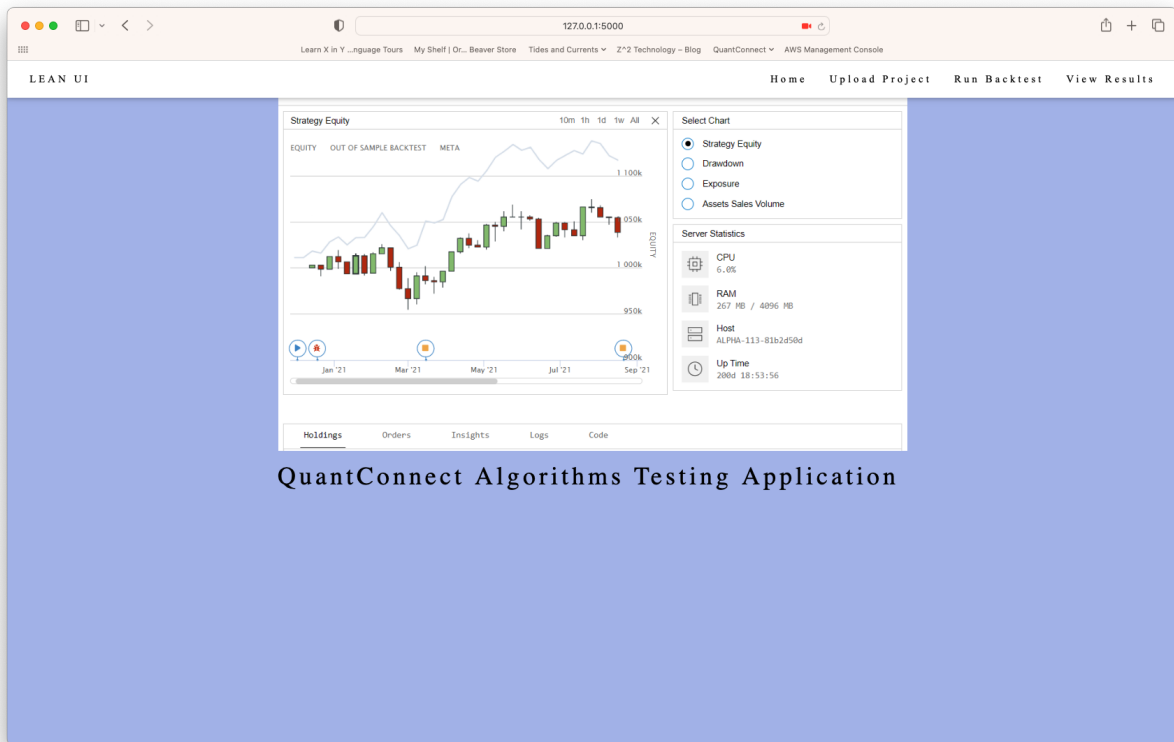
#####
#                                                                    #
#                                                                    #
#                               Helpers                               #
#                                                                    #
#                                                                    #
#####

def check_unique_project_backtest(project_name=None, backtest_name=None):...
```

A general overview of the project's route handlers. Users start at "load_project" which asks the user to upload an algorithm and give it a unique name. From there the app sends a POST request to create_project where the application uses Lean's API to create a new project and store it in the file structure and upload the details to the database. From there, the user can send a GET request to "run_project" which will display the projects available to backtest, ask for a unique backtest name and any parameters the user wishes to modify, and then sends a POST request back to run_project. In the POST request of run_project, the application will invoke the Lean API to run the backtest against any defined parameters, create the necessary file paths, and upload the data to the database after creating a new Backtest Object. The user can then send a GET request to select_backtest which will provide a drop down of backtests the user can see the results and plots for. The POST request to select_backtest will grab the information pertaining to that backtest from the database and display the results on the results page.

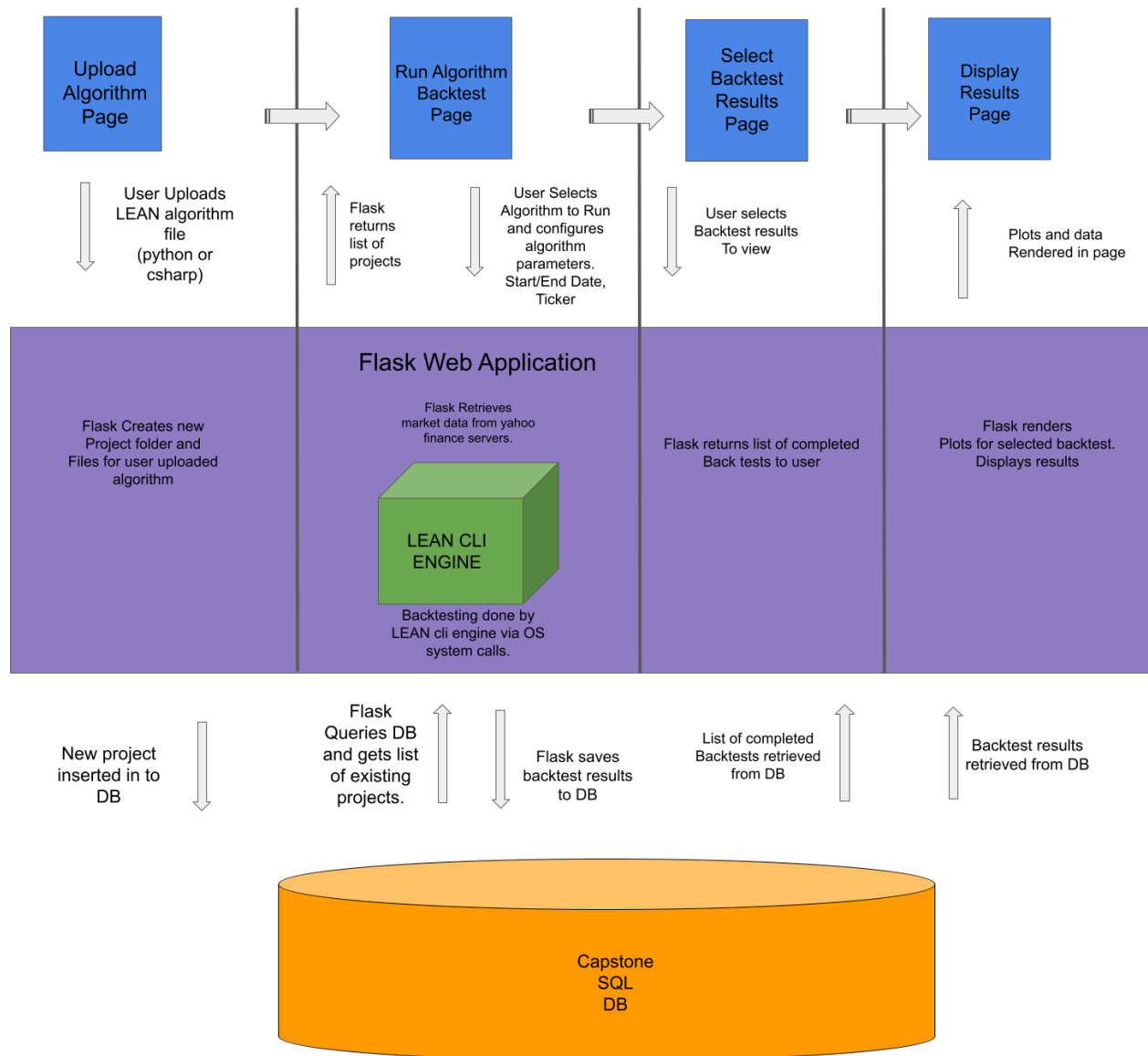


Snippet from the Results page.



QuantConnect Algorithms Testing Application

Application landing page



Outline of app workflow.

```

@app.route('/project-selector', methods=['GET', 'POST'])
def run_project():

    db_conn = Database()
    db_conn.connect_to_database()

    # Select project from list
    if request.method == 'GET':

        # Get project file paths stored in database to populate dropdown menu
        stored_project_names_ids = db_conn.select_project_names_ids_query()
        return render_template('select_project.html',
                               projects=stored_project_names_ids)

    # Run project
    elif request.method == 'POST':

        # gets selected project name from select_project.html
        project_id = request.form.get("projects")
        # build the --output path for running backtest
        backtest_name = request.form.get("backtest_destination")
        output_path = dirname + app.config[
            'BACKTEST_FOLDER'] + backtest_name

        # Get the file paths of selected project with matching project_id
        stored_project_file_paths = db_conn.select_project_paths_query(
            project_id)
        project_root_path = stored_project_file_paths[0][0]
        project_config_path = stored_project_file_paths[0][1]

        # Check if backtest with same name already exists in database
        unique_backtest_count = check_unique_project_backtest(
            backtest_name=backtest_name)
        if unique_backtest_count != 0:
            # backtest name already exists
            return render_template('select_project.html',
                                   error="Enter a unique backtest name. "
                                   "A backtest with that name already "
                                   "exists in the database")

        # Load data if requested
        tickers = request.form.get("tickers_to_load")
        start_date = request.form.get("start_date")
        end_date = request.form.get("end_date")
        overwrite_data = request.form.get("overwrite_existing_data")
        configuration_parameters = {
            'ticker': tickers,
            'start_date': start_date,
            'end_date': end_date
        }

        if overwrite_data is not None:
            overwrite_data = True

```

```

if tickers != "":
    tickers = tickers.split(",")
    get_yahoo_data(tickers, start_date, end_date, overwrite_data)

# Create backtest object
backtest_object = Backtest(backtest_name=backtest_name,
                            backtest_filename=output_path,
                            project_id=project_id,
                            algorithm_config_filepath=project_config_path,
                            configuration_parameters=configuration_parameters)

backtest_object.run_backtest(project_root_path)

# Find the two json files in the output path
for file in os.listdir(output_path):
    file_ext = os.path.splitext(file)
    if (file_ext[1] == '.json'
        and 'data-monitor-report' not in file_ext[0]):

        # json backtest order events file
        if '-order-events' in file_ext[0]:
            backtest_orders_file_path = os.path.join(output_path,
                                                    file)
            backtest_object.set_backtest_order_events_filename(
                backtest_orders_file_path)
        # regular backtest file
        else:
            backtest_file_path = os.path.join(output_path,
                                              file)
            backtest_object.set_backtest_filename(backtest_file_path)

backtest_object.insert_backtest_query()

# Get backtest object back for plotting
backtest_object.select_backtest_from_db()

# Create plots
filename = plot_equity(backtest_object)
plot = Plot(backtest_id=backtest_object.backtest_id,
            plot_type="plot_equity", plot_file_name=filename)
plot.create_plot()

return redirect('/select-backtest')

```