In [3]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
import folium
from scipy.stats import pearsonr
from folium import plugins
from folium.plugins import HeatMap
from folium.plugins import HeatMapWithTime
import calendar
from matplotlib.ticker import FuncFormatter
import datetime
import joblib
import xgboost as xgb
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import train_test_split
import statsmodels
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

In [4]:

```
data = pd.read_excel('consumption.xlsx')
data.head()
```

Out[4]:

	Month	Year	Day of the Month	Rain fall,Inches	TotalTreated water leaving plant,MG	Hours Plant in operation	Total Raw water to plant,MG	Compliance with permitted capacity?	Backwas,Thousand Gallons	Peak demand into distribution,MDG	...	Measuremnt Recorded ≥ Measuremnt Required?	№ SpecifiedTreatn
0	January	2017	1	0	15.38	24	27.14	yes	3043	15.29	...	Yes	
1	January	2017	2	0	15.56	24	27.13	yes	3076	15.47	...	Yes	
2	January	2017	3	0	15.74	24	27.14	yes	3110	15.6	...	Yes	
3	January	2017	4	0	16.02	24	26.83	yes	3095	15.62	...	Yes	
4	January	2017	5	0	15.25	24	26.63	yes	3038	15.5	...	Yes	

5 rows × 22 columns

In [5]:

```
data1 = pd.read_csv('Cleaned.csv')
data1.head()
```

Out[5]:

	Date	Adjusted Daily Consumption, Kwh
0	01/01/2017	153910
1	02/01/2017	153910
2	03/01/2017	153910
3	04/01/2017	153910
4	05/01/2017	153910

In [6]:

```
column_names = data.columns.tolist()

# Print the list of column names
print("List of columns in the DataFrame:")
print(column_names)
```

```
List of columns in the DataFrame:
['Month', 'Year', 'Day of the Month', 'Rain fall,Inches', 'TotalTreated water leaving plant,MG', 'Hours Plant in operatio
n', 'Total Raw water to plant,MG', 'Compliance with permitted capacity?', 'Backwas,Thousand Gallons', 'Peak demand into dis
tribution,MDG', 'Nomeasurements Recorded', 'No Measuremnts Required', 'Measuremnt Recorded ≥ Measuremnt Required?', '№ Meas
uremnt ≤ SpecifiedTreatmentTeachique Limit', '№ Measuremmt > Never Exceed Limit', 'Average Daily Turbidity,NTU', 'Maximum D
ailyTurbidity,NTU', 'Log Inactivation,Giardia', 'Giardia Compliance?', 'Log Inactivation,Viruses', 'Virus Compliance?', 'Em
ergency or Abnormal Operating Conditions,Repairor Maintenance Work that InvolvesTaking Water System Components Out of Opera
tion']
```

In [7]:

```
# List of columns to check for non-numeric values
columns_to_check = ['Rain fall,Inches', 'TotalTreated water leaving plant,MG',  'Hours Plant in operation', 'Total Raw water to plant
                    'Backwas,Thousand Gallons', 'Peak demand into distribution,MDG', 'Nomeasurements Recorded',
                    'No Measuremnts Required',  'Average Daily Turbidity,NTU', 'Maximum DailyTurbidity,NTU', 'Log Inactivation,Giardi
                     'Log Inactivation,Viruses']
# Convert to numeric, coercing non-numeric to NaN
for col in columns_to_check:
    data[col] = pd.to_numeric(data[col], errors='coerce')

# Drop rows where any of the specified columns are NaN
data.dropna(subset=columns_to_check, inplace=True)

# Reset the index
data.reset_index(drop=True, inplace=True)
```

In [8]:

```
column_data_types = data.dtypes

# Print the data types
print("Data types of each column in the DataFrame:")
print(column_data_types)
```

```
Data types of each column in the DataFrame:
Month
object
Year
int64
Day of the Month
object
Rain fall,Inches
float64
TotalTreated water leaving plant,MG
float64
Hours Plant in operation
float64
Total Raw water to plant,MG
float64
Compliance with permitted capacity?
object
Backwas,Thousand Gallons
float64
Peak demand into distribution,MDG
float64
Nomeasurements Recorded
float64
No Measuremnts Required
float64
Measuremnt Recorded ≥ Measuremnt Required?
object
№ Measuremnt ≤ SpecifiedTreatmentTeachique Limit
object
№ Measuremmt > Never Exceed Limit
object
Average Daily Turbidity,NTU
float64
Maximum DailyTurbidity,NTU
float64
Log Inactivation,Giardia
float64
Giardia Compliance?
object
Log Inactivation,Viruses
float64
Virus Compliance?
object
Emergency or Abnormal Operating Conditions,Repairor Maintenance Work that InvolvesTaking Water System Components Out of Ope
ration     object
dtype: object
```

In [9]:

```
# Correct the year naming issue (changing "20222" to "2022")
data['Year'] = data['Year'].replace(20222, 2022)

# Convert Month, Day, and Year columns to a datetime format
data['Date'] = pd.to_datetime(data['Year'].astype(str) + '-' + data['Month'].astype(str) + '-' + data['Day of the Month'].astype(str)

```

In [10]:

```
# Extract the required columns
data = data[['Date',  'Year', 'Month', 'Day of the Month', 'Rain fall,Inches', 'TotalTreated water leaving plant,MG',  'Hours Plant i
                   'Backwas,Thousand Gallons', 'Peak demand into distribution,MDG', 'Nomeasurements Recorded',
                   'No Measuremnts Required',  'Average Daily Turbidity,NTU', 'Maximum DailyTurbidity,NTU', 'Log Inactivation,Giardi
                    'Log Inactivation,Viruses']]

# Display the extracted data
data.head()
```

Out[10]:

	Date	Year	Month	Day of the Month	Rain fall,Inches	TotalTreated water leaving plant,MG	Hours Plant in operation	Total Raw water to plant,MG	Backwas,Thousand Gallons	Peak demand into distribution,MDG	Nomeasurements Recorded	No Measuremnts Required	Turbi
0	2017-01-01	2017	January	1	0.0	15.38	24.0	27.14	3043.0	15.29	6.0	6.0	
1	2017-01-02	2017	January	2	0.0	15.56	24.0	27.13	3076.0	15.47	6.0	6.0	
2	2017-01-03	2017	January	3	0.0	15.74	24.0	27.14	3110.0	15.60	6.0	6.0	
3	2017-01-04	2017	January	4	0.0	16.02	24.0	26.83	3095.0	15.62	6.0	6.0	
4	2017-01-05	2017	January	5	0.0	15.25	24.0	26.63	3038.0	15.50	6.0	6.0	

In [11]:

```
# Convert the 'Date' column in data1 to datetime format
data1['Date'] = pd.to_datetime(data1['Date'])

# Now you can perform the merge
data = pd.merge(data, data1, on='Date', how='inner')

```

```
C:\Users\User\AppData\Local\Temp\ipykernel_18284\3107944606.py:2: UserWarning: Parsing '13/01/2017' in DD/MM/YYYY format.
Provide format or specify infer_datetime_format=True for consistent parsing.
  data1['Date'] = pd.to_datetime(data1['Date'])
C:\Users\User\AppData\Local\Temp\ipykernel_18284\3107944606.py:2: UserWarning: Parsing '14/01/2017' in DD/MM/YYYY format.
Provide format or specify infer_datetime_format=True for consistent parsing.
  data1['Date'] = pd.to_datetime(data1['Date'])
C:\Users\User\AppData\Local\Temp\ipykernel_18284\3107944606.py:2: UserWarning: Parsing '15/01/2017' in DD/MM/YYYY format.
Provide format or specify infer_datetime_format=True for consistent parsing.
  data1['Date'] = pd.to_datetime(data1['Date'])
C:\Users\User\AppData\Local\Temp\ipykernel_18284\3107944606.py:2: UserWarning: Parsing '16/01/2017' in DD/MM/YYYY format.
Provide format or specify infer_datetime_format=True for consistent parsing.
  data1['Date'] = pd.to_datetime(data1['Date'])
C:\Users\User\AppData\Local\Temp\ipykernel_18284\3107944606.py:2: UserWarning: Parsing '17/01/2017' in DD/MM/YYYY format.
Provide format or specify infer_datetime_format=True for consistent parsing.
  data1['Date'] = pd.to_datetime(data1['Date'])
C:\Users\User\AppData\Local\Temp\ipykernel_18284\3107944606.py:2: UserWarning: Parsing '18/01/2017' in DD/MM/YYYY format.
Provide format or specify infer_datetime_format=True for consistent parsing.
  data1['Date'] = pd.to_datetime(data1['Date'])
C:\Users\User\AppData\Local\Temp\ipykernel_18284\3107944606.py:2: UserWarning: Parsing '19/01/2017' in DD/MM/YYYY format.
Provide format or specify infer_datetime_format=True for consistent parsing.
```

In [12]:

```
column_data_types = data.dtypes

# Print the data types
print("Data types of each column in the DataFrame:")
print(column_data_types)
```

```
Data types of each column in the DataFrame:
Date                                       datetime64[ns]
Year                                                int64
Month                                              object
Day of the Month                                   object
Rain fall,Inches                                  float64
TotalTreated water leaving plant,MG               float64
Hours Plant in operation                          float64
Total Raw water to plant,MG                       float64
Backwas,Thousand Gallons                          float64
Peak demand into distribution,MDG                 float64
Nomeasurements Recorded                           float64
No Measuremnts Required                           float64
Average Daily Turbidity,NTU                       float64
Maximum DailyTurbidity,NTU                        float64
Log Inactivation,Giardia                          float64
Log Inactivation,Viruses                          float64
Adjusted Daily Consumption, Kwh                     int64
dtype: object
```

In [14]:

```
data = pd.get_dummies(data, columns=['Month'], drop_first=True)

# Convert 'Day of the Month' to integer if it's an object
if data['Day of the Month'].dtype == 'object':
    data['Day of the Month'] = data['Day of the Month'].astype(int)

```

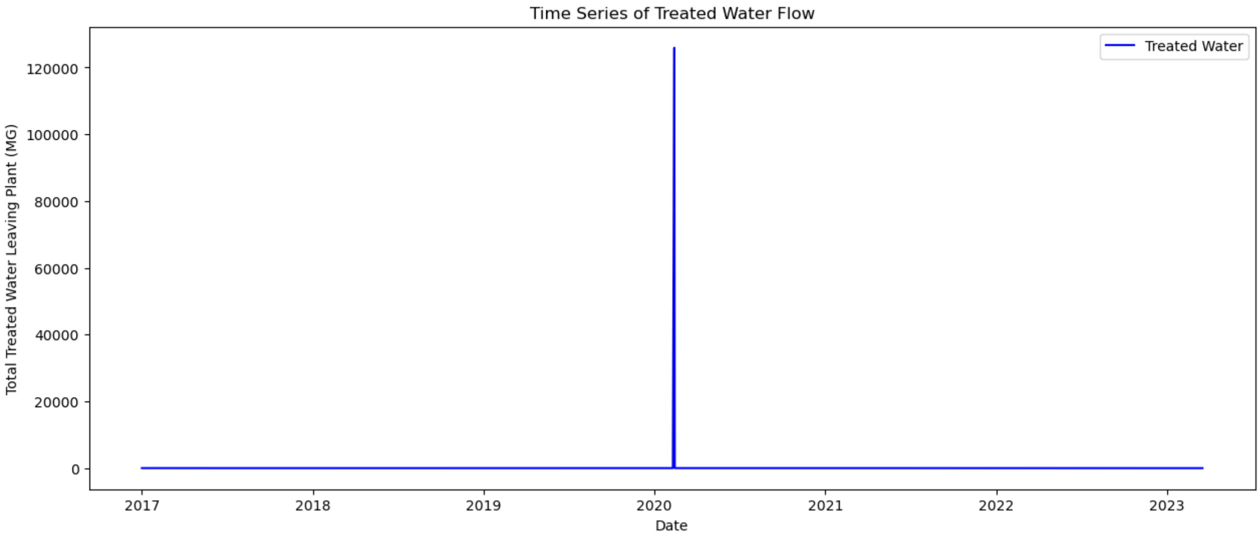
In [15]:

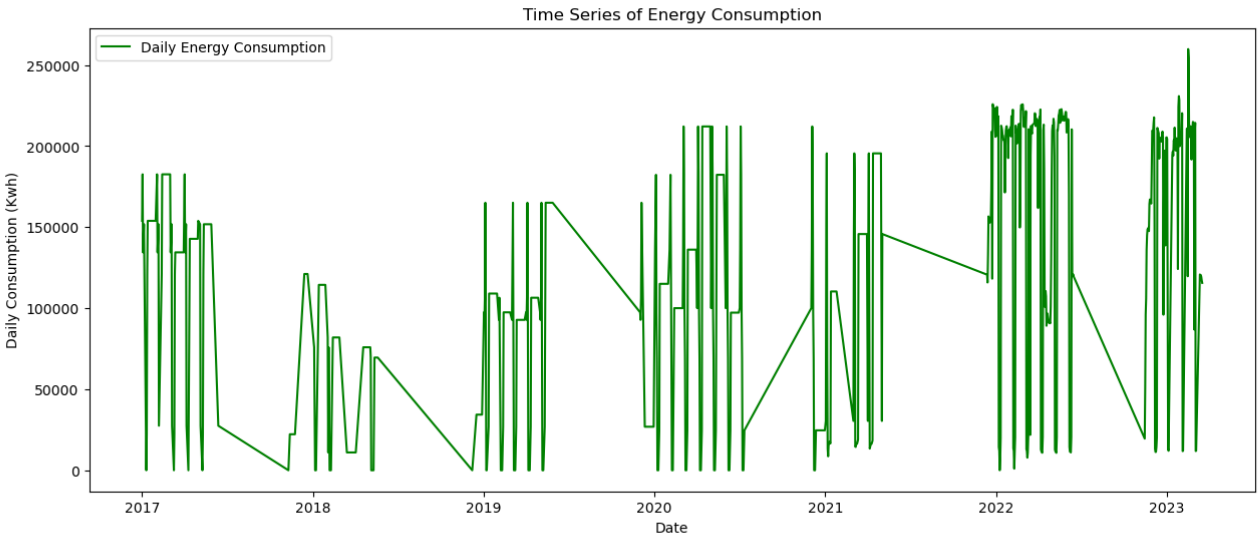
```
# Check the data types of each column
print('Data types of each column in the DataFrame:')
print(data.dtypes)

# Check for missing values
print('\nMissing values in each column:')
print(data.isnull().sum())

# Display basic statistics
print('\nBasic statistics:')
print(data.describe())

```

```
count      877.000000  877.000000      877.000000     877.000000  877.000000
mean         0.119726    0.045610        0.099202       0.196123    0.011403
std          0.324826    0.208757        0.299103       0.397289    0.106233
min          0.000000    0.000000        0.000000       0.000000    0.000000
25%          0.000000    0.000000        0.000000       0.000000    0.000000
50%          0.000000    0.000000        0.000000       0.000000    0.000000
75%          0.000000    0.000000        0.000000       0.000000    0.000000
max          1.000000    1.000000        1.000000       1.000000    1.000000

       Month_June  Month_March   Month_May  Month_November
count  877.000000   877.000000  877.000000      877.000000
mean     0.051311     0.174458    0.119726        0.027366
std      0.220758     0.379720    0.324826        0.163241
min      0.000000     0.000000    0.000000        0.000000
25%      0.000000     0.000000    0.000000        0.000000
50%      0.000000     0.000000    0.000000        0.000000
75%      0.000000     0.000000    0.000000        0.000000
max      1.000000     1.000000    1.000000        1.000000

[8 rows x 24 columns]
```

In [16]:

```
# Plot for Total Treated Water Leaving Plant
plt.figure(figsize=(15, 6))
plt.plot(data['Date'], data['TotalTreated water leaving plant,MG'], label='Treated Water', color='blue')
plt.xlabel('Date')
plt.ylabel('Total Treated Water Leaving Plant (MG)')
plt.title('Time Series of Treated Water Flow')
plt.legend()
plt.show()

# Plot for Total Raw Water to Plant
plt.figure(figsize=(15, 6))
plt.plot(data['Date'], data['Adjusted Daily Consumption, Kwh'], label='Daily Energy Consumption', color='green')
plt.xlabel('Date')
plt.ylabel('Daily Consumption (Kwh)')
plt.title('Time Series of Energy Consumption')
plt.legend()
plt.show()

```





In [17]:

```
# Removing outliers beyond 99th percentile
for col in ['TotalTreated water leaving plant,MG', 'Total Raw water to plant,MG', 'Peak demand into distribution,MDG']:
    cap = data[col].quantile(0.99)
    data = data[data[col] <= cap]
```

In [18]:

```
data.head()
```

Out[18]:

	Date	Year	Day of the Month	Rain fall,Inches	TotalTreated water leaving plant,MG	Hours Plant in operation	Total Raw water to plant,MG	Backwas,Thousand Gallons	Peak demand into distribution,MDG	Nomeasurements Recorded	...	Adjusted Daily Consumption, Kwh	Month_De
0	2017-01-01	2017	1	0.0	15.38	24.0	27.14	3043.0	15.29	6.0	...	153910	
1	2017-01-02	2017	2	0.0	15.56	24.0	27.13	3076.0	15.47	6.0	...	182614	
2	2017-01-03	2017	3	0.0	15.74	24.0	27.14	3110.0	15.60	6.0	...	134468	
3	2017-01-04	2017	4	0.0	16.02	24.0	26.83	3095.0	15.62	6.0	...	142808	
4	2017-01-05	2017	5	0.0	15.25	24.0	26.63	3038.0	15.50	6.0	...	151815	

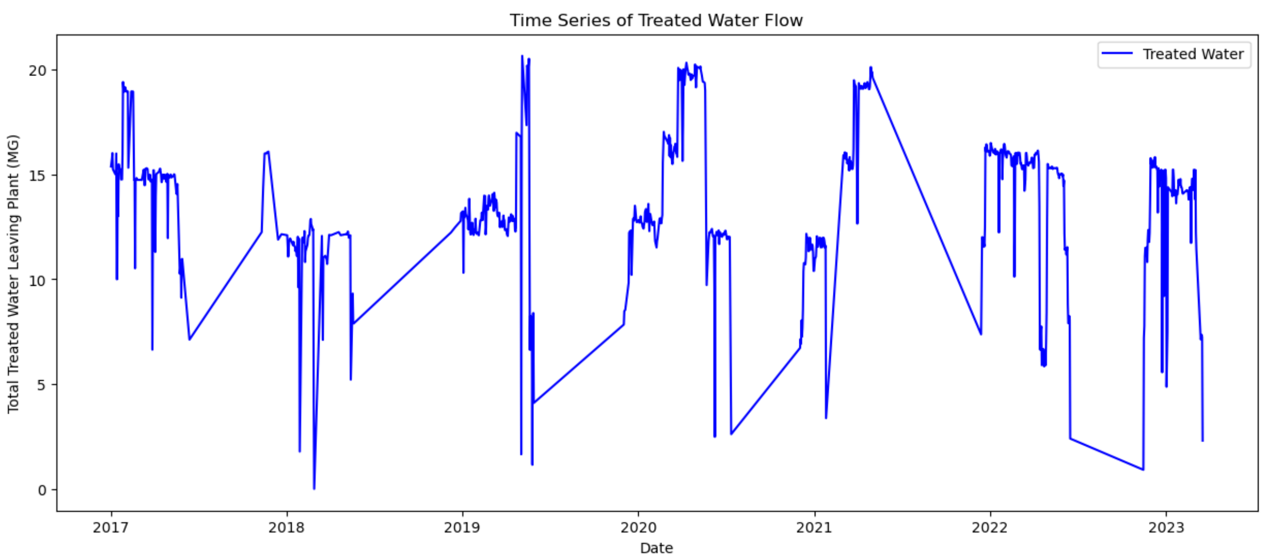
5 rows × 25 columns

In [19]:

```
# Checking for non-numeric values

# Function to check if a string contains only digits and periods
def check_string(value):
    for char in str(value):
        if char not in '1234567890.':
            return False
    return True

# Dictionary to store columns with non-integer or non-period characters
columns_with_non_integers = {}

# Loop through each column that is supposed to be numeric after preprocessing
for col in data:  # Replace with your actual numeric columns
    if col != "Date":
      # Check each value in the column
      for value in data[col]:
          if not check_string(value):
              if col not in columns_with_non_integers:
                  columns_with_non_integers[col] = []
              columns_with_non_integers[col].append(value)

# Display the columns and their non-integer values
for col, values in columns_with_non_integers.items():
    print(f"Column '{col}' has non-integer values: {values[:10]}...")  # Display only the first 10 non-integer values
```

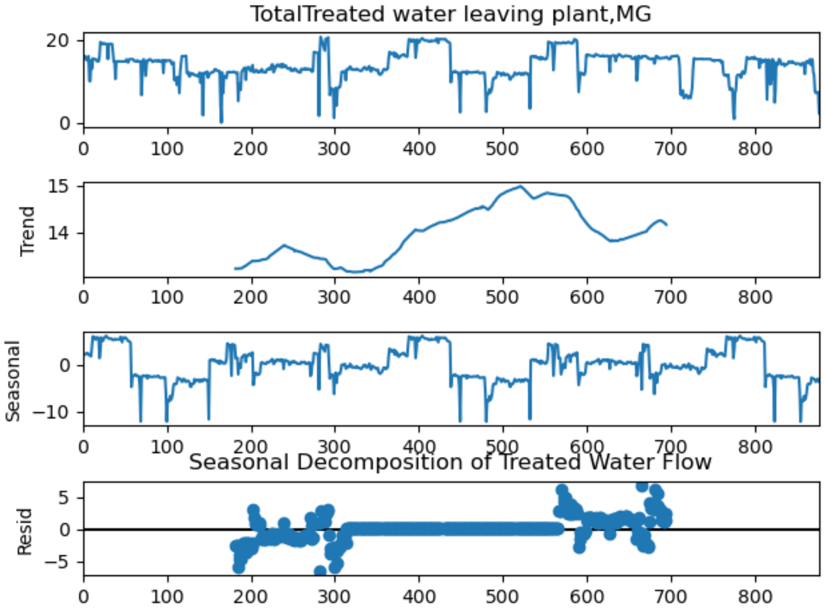
In [20]:

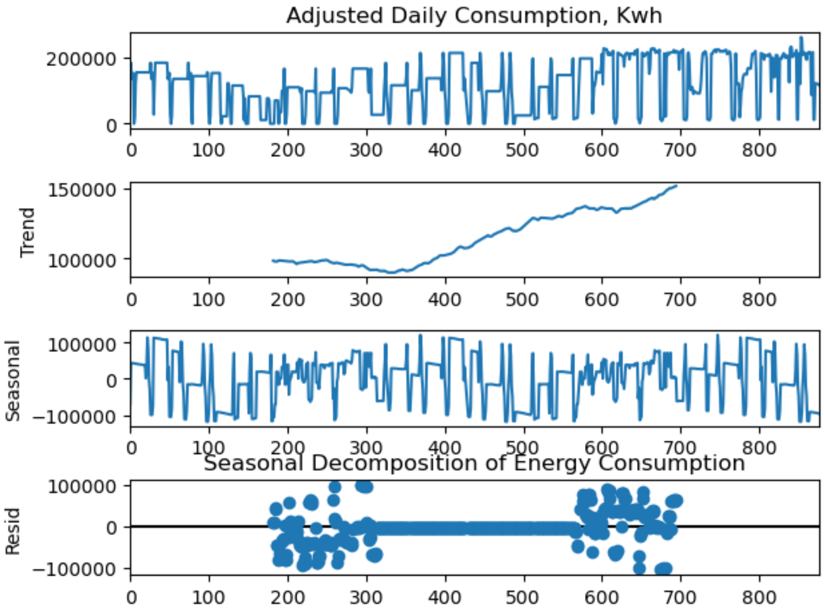
```
# Plot for Total Treated Water Leaving Plant
plt.figure(figsize=(15, 6))
plt.plot(data['Date'], data['TotalTreated water leaving plant,MG'], label='Treated Water', color='blue')
plt.xlabel('Date')
plt.ylabel('Total Treated Water Leaving Plant (MG)')
plt.title('Time Series of Treated Water Flow')
plt.legend()
plt.show()

```



In [21]:

```
# Seasonal Decomposition for Treated Water
decomposition_treated = seasonal_decompose(data['TotalTreated water leaving plant,MG'], period=365)
decomposition_treated.plot()
plt.title('Seasonal Decomposition of Treated Water Flow')
plt.show()

# Seasonal Decomposition for Raw Water
decomposition_raw = seasonal_decompose(data['Adjusted Daily Consumption, Kwh'], period=365)
decomposition_raw.plot()
plt.title('Seasonal Decomposition of Energy Consumption')
plt.show()

```





In [22]:

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Perform seasonal decomposition for Treated Water

treated_seasonal = decomposition_treated.seasonal
treated_trend = decomposition_treated.trend
treated_residual = decomposition_treated.resid

# Perform seasonal decomposition for Raw Water
raw_seasonal = decomposition_raw.seasonal
raw_trend = decomposition_raw.trend
raw_residual = decomposition_raw.resid

# Create DataFrames to hold the decomposition components for each target variable
decomposition_treated_df = pd.DataFrame({
    'Date': data['Date'],
    'Seasonal': treated_seasonal,
    'Trend': treated_trend,
    'Residual': treated_residual
})

decomposition_raw_df = pd.DataFrame({
    'Date': data['Date'],
    'Seasonal': raw_seasonal,
    'Trend': raw_trend,
    'Residual': raw_residual
})

# Display the first few rows of each DataFrame
print("Decomposition Components for Treated Water:")
print(decomposition_treated_df.head())

print("\nDecomposition Components for Energy Consumption:")
print(decomposition_raw_df.head())



# Create DataFrames to hold the decomposition components for each target variable
decomposition_treated_df = pd.DataFrame({
    'Date': data['Date'],
    'Seasonal': treated_seasonal,
    'Trend': treated_trend,
    'Residual': treated_residual
})

decomposition_raw_df = pd.DataFrame({
    'Date': data['Date'],
    'Seasonal': raw_seasonal,
    'Trend': raw_trend,
    'Residual': raw_residual
})

# Display the first few non-NaN rows of each DataFrame
print("Decomposition Components for Treated Water:")
print(decomposition_treated_df.dropna().head())

print("\nDecomposition Components for Daily Consumption, Kwh:")
print(decomposition_raw_df.dropna().head())

```

```
Decomposition Components for Treated Water:
        Date  Seasonal  Trend  Residual
0 2017-01-01  1.664258    NaN       NaN
1 2017-01-02  1.886807    NaN       NaN
2 2017-01-03  2.157730    NaN       NaN
3 2017-01-04  2.225113    NaN       NaN
4 2017-01-05  2.376322    NaN       NaN

Decomposition Components for Energy Consumption:
        Date      Seasonal  Trend  Residual
0 2017-01-01 -69051.966857    NaN       NaN
1 2017-01-02  42405.934513    NaN       NaN
2 2017-01-03  42551.756431    NaN       NaN
3 2017-01-04  42692.142732    NaN       NaN
4 2017-01-05  42346.934513    NaN       NaN
Decomposition Components for Treated Water:
          Date  Seasonal      Trend  Residual
182 2018-05-10  1.345421  13.227793 -2.593213
183 2018-05-11  1.428341  13.228119 -2.606460
184 2018-05-13  1.392558  13.228615 -2.521173
185 2018-05-14 -2.058347  13.229413 -5.961067
186 2018-05-18  0.132310  13.229733 -4.042043

Decomposition Components for Daily Consumption, Kwh:
          Date       Seasonal         Trend      Residual
182 2018-05-10 -105600.116172  98094.030137   7506.086035
183 2018-05-11 -105532.798364  97739.865753   7792.932610
184 2018-05-13  -70695.391514  97612.545205  42627.846309
185 2018-05-14  -69819.966857  97617.131507  41747.835350
186 2018-05-18   18693.646842  97598.868493 -46747.515335
```

In [23]:

```python
from statsmodels.tsa.stattools import adfuller

# Perform Augmented Dickey-Fuller test
def adf_test(series):
    result = adfuller(series, autolag='AIC')
    print(f'ADF Statistic: {result[0]}')
    print(f'p-value: {result[1]}')
    print(f'Critical Values: {result[4]}')
    if result[1] <= 0.05:
        print('Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root and is stationary')
    else:
        print('Weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary ')

# Apply the test on your time series data
adf_test(data['TotalTreated water leaving plant,MG'])
adf_test(data['Adjusted Daily Consumption, Kwh'])

```

```
ADF Statistic: -4.533456769576553
p-value: 0.00017100684509673158
Critical Values: {'1%': -3.4381216826257956, '5%': -2.8649705364894635, '10%': -2.568596692178972}
Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root and is stationary
ADF Statistic: -6.58512382505417
p-value: 7.3474096452009324e-09
Critical Values: {'1%': -3.4381032536542913, '5%': -2.8649624121419746, '10%': -2.5685923644574107}
Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root and is stationary
```

In [24]:

```python
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt

# Train-Test Split
data = data.sort_values('Date')
train_size = int(len(data) * 0.8)
train, test = data[0:train_size], data[train_size:len(data)]
```

In [25]:

```
# SARIMA Model Training for Treated Water
sarima_model_treated = SARIMAX(train['TotalTreated water leaving plant,MG'],
                               order=(1, 1, 1),
                               seasonal_order=(1, 1, 1, 12))
sarima_fit_treated = sarima_model_treated.fit(disp=False)

# SARIMA Model Training for Raw Water
sarima_model_raw = SARIMAX(train['Adjusted Daily Consumption, Kwh'],
                           order=(1, 1, 1),
                           seasonal_order=(1, 1, 1, 12))
sarima_fit_raw = sarima_model_raw.fit(disp=False)

```

```
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
```

In [26]:

```
# Model Evaluation
# SARIMA
sarima_pred_treated = sarima_fit_treated.predict(len(train), len(data)-1)
sarima_pred_raw = sarima_fit_raw.predict(len(train), len(data)-1)
```

```
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:834: ValueWarning: No supported index is availa
ble. Prediction results will be given with an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:834: ValueWarning: No supported index is availa
ble. Prediction results will be given with an integer index beginning at `start`.
  return get_prediction_index(
```

In [27]:

```
# Calculate RMSE and MAE
print("SARIMA RMSE for Treated Water:", sqrt(mean_squared_error(test['TotalTreated water leaving plant,MG'], sarima_pred_treated)))
print("SARIMA RMSE for Energy Consumption:", sqrt(mean_squared_error(test['Adjusted Daily Consumption, Kwh'], sarima_pred_raw)))
```

```
SARIMA RMSE for Treated Water: 4.7015497521295115
SARIMA RMSE for Energy Consumption: 72614.11760287227
```

In [28]:

```
print("SARIMA MAE for Treated Water:", mean_absolute_error(test['TotalTreated water leaving plant,MG'], sarima_pred_treated))
print("SARIMA MAE for Energy Consumption:", mean_absolute_error(test['Adjusted Daily Consumption, Kwh'], sarima_pred_raw))

#print("Prophet MAE for Treated Water:", mean_absolute_error(test['TotalTreated water leaving plant,MG'], prophet_pred_treated))
#print("Prophet MAE for Raw Water:", mean_absolute_error(test['Total Raw water to plant,MG'], prophet_pred_raw))
```

```
SARIMA MAE for Treated Water: 3.2650915054531637
SARIMA MAE for Energy Consumption: 53888.83678841617
```

Random Forest as the machine learning model and SARIMA as the time series model. It then averages the predictions from both models to create an ensemble. You can adjust the ensemble weights based on the performance of individual models.

In [29]:

```
# Importing required libraries
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.model_selection import train_test_split
from statsmodels.tsa.statespace.sarimax import SARIMAX
from math import sqrt
```

In [30]:

```
# Assuming 'data' is your DataFrame containing historical data
# Log transformation of the target variables
data['Log_TotalTreated'] = np.log1p(data['TotalTreated water leaving plant,MG'])
data['Log_EnergyConsumption'] = np.log1p(data['Adjusted Daily Consumption, Kwh'])

# Feature Engineering: Create lag features for target variables
for col in ['Log_EnergyConsumption']:
    for lag in range(1, 4):  # Create 3 lag features
        data[f'{col}_lag{lag}'] = data[col].shift(lag)

# Drop NaN values created due to lag features
data.dropna(inplace=True)
```

In [122]:

```
# Split sizes
train_size = int(len(data) * 0.6)  # 70% for training
validation_size = int(len(data) * 0.2)  # 20% for validation

# Train-Validation-Test Split
train, temp = data[:train_size], data[train_size:]
validation, test = temp[:validation_size], temp[validation_size:]

# Prepare data for machine learning models for actual values

# Training data
X_train_actual = train.drop(['Date', 'TotalTreated water leaving plant,MG', 'Adjusted Daily Consumption, Kwh'], axis=1)
y_train_treated_actual = train['TotalTreated water leaving plant,MG']
y_train_energy_actual = train['Adjusted Daily Consumption, Kwh']

# Validation data
X_validation_actual = validation.drop(['Date', 'TotalTreated water leaving plant,MG', 'Adjusted Daily Consumption, Kwh'], axis=1)
y_validation_treated_actual = validation['TotalTreated water leaving plant,MG']
y_validation_energy_actual = validation['Adjusted Daily Consumption, Kwh']

# Test data
X_test_actual = test.drop(['Date', 'TotalTreated water leaving plant,MG', 'Adjusted Daily Consumption, Kwh'], axis=1)
y_test_treated_actual = test['TotalTreated water leaving plant,MG']
y_test_energy_actual = test['Adjusted Daily Consumption, Kwh']

print("Training set size:", len(X_train_actual))
print("Validation set size:", len(X_validation_actual))
print("Test set size:", len(X_test_actual))

```

```
Training set size: 510
Validation set size: 170
Test set size: 170
```

In [68]:

```
X_train_actual
```

Out[68]:

	Year	Day of the Month	Rain fall,Inches	Hours Plant in operation	Total Raw water to plant,MG	Backwas,Thousand Gallons	Peak demand into distribution,MDG	Nomeasurements Recorded	No Measuremnts Required	Average Daily Turbidity,NTU	...	Month_July
3	2017	4	0.0	24.0	26.830000	3095.0000	15.6200	6.0	6.0	0.0200	...	0
4	2017	5	0.0	24.0	26.630000	3038.0000	15.5000	6.0	6.0	0.0200	...	0
5	2017	10	0.0	24.0	26.900000	3045.0000	15.6400	6.0	6.0	0.0200	...	0
6	2017	11	0.0	24.0	26.530000	2982.0000	15.5700	6.0	6.0	0.0000	...	0
7	2017	12	0.0	24.0	26.320000	2951.0000	15.3300	6.0	6.0	0.0200	...	0
...	...	...	...	...	...	...	...	...	...	...	...	...
616	2022	8	0.0	24.0	29.462891	3245.9990	16.2395	6.0	6.0	0.0722	...	0
617	2022	9	0.0	24.0	18.392578	3311.4417	16.2369	6.0	6.0	0.0722	...	0
618	2022	10	0.0	24.0	20.433594	3280.6940	16.2481	6.0	6.0	0.0719	...	0
619	2022	11	0.0	24.0	22.681641	3158.2337	19.4189	6.0	6.0	0.0723	...	0
620	2022	12	0.0	24.0	17.880859	3193.6305	16.2808	6.0	6.0	0.0756	...	0

595 rows × 27 columns

In [150]:

```

# Machine Learning Model Training for Treated Water
rf_model_treated = RandomForestRegressor()
rf_model_treated.fit(X_train_actual, y_train_treated_actual)

gb_model_treated = GradientBoostingRegressor()
gb_model_treated.fit(X_train_actual, y_train_treated_actual)

# Machine Learning Model Training for Energy Consumption
rf_model_energy = RandomForestRegressor()
rf_model_energy.fit(X_train_actual, y_train_energy_actual)

gb_model_energy = GradientBoostingRegressor()
gb_model_energy.fit(X_train_actual, y_train_energy_actual)

# SARIMA Model Training for Treated Water
try:
    sarima_model_treated = SARIMAX(y_train_treated_actual, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
    sarima_fit_treated = sarima_model_treated.fit(disp=False)
except Exception as e:
    print("Error fitting SARIMA model for Treated Water:", str(e))

# SARIMA Model Training for Energy Consumption
try:
    sarima_model_energy = SARIMAX(y_train_energy_actual, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
    sarima_fit_energy = sarima_model_energy.fit(disp=False)
except Exception as e:
    print("Error fitting SARIMA model for Energy Consumption:", str(e))
```

```
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:471: ValueWarning: An unsupported index was pro
vided and will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
```

In [149]:

```
# Making Predictions for Actual Values on Validation Set
rf_pred_treated_validation = rf_model_treated.predict(X_validation_actual)
gb_pred_treated_validation = gb_model_treated.predict(X_validation_actual)
rf_pred_energy_validation = rf_model_energy.predict(X_validation_actual)
gb_pred_energy_validation = gb_model_energy.predict(X_validation_actual)

sarima_pred_treated_validation = sarima_fit_treated.predict(len(y_train_treated_actual), len(y_train_treated_actual) + len(y_validati
sarima_pred_energy_validation = sarima_fit_energy.predict(len(y_train_energy_actual), len(y_train_energy_actual) + len(y_validation_e

# Convert SARIMA predictions to numpy arrays for consistency
sarima_pred_treated_validation = np.array(sarima_pred_treated_validation)
sarima_pred_energy_validation = np.array(sarima_pred_energy_validation)

# Ensemble Predictions for Actual Values on Validation Set
treated_validation_outputs = [rf_pred_treated_validation, gb_pred_treated_validation, sarima_pred_treated_validation]
energy_validation_outputs = [rf_pred_energy_validation, gb_pred_energy_validation, sarima_pred_energy_validation]
ensemble_pred_treated_validation = np.mean(treated_validation_outputs, axis=0)
ensemble_pred_energy_validation = np.mean(energy_validation_outputs, axis=0)


```

```
C:\Users\User\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:834: ValueWarning: No supported index is availa
ble. Prediction results will be given with an integer index beginning at `start`.
  return get_prediction_index(
```

In [125]:

```
def error(y_true, y_pred, threshold=0.5):
    """
    Calculate the error rate based on a given threshold.
    The function returns the proportion of predictions that are off by more than the threshold.
    """
    return np.mean(np.abs(y_true - y_pred) > threshold)
```

In [126]:

```
from sklearn.metrics import mean_squared_error, mean_absolute_error
from math import sqrt

def calculate_metrics(y_true, y_pred, model_name, target_name):
    rmse = sqrt(mean_squared_error(y_true, y_pred))
    mae = mean_absolute_error(y_true, y_pred)
    error_rate = calculate_error_rate(y_true, y_pred)  # Define this function!

    print(f"=== {model_name} for {target_name} ===")
    print(f"RMSE: {rmse:.2f}")
    print(f"MAE: {mae:.2f}")
    print(f"Error Rate: {error_rate:.2f}%\n")

# For example: calculate_error_rate can be a relative error, defined as:
def calculate_error_rate(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100  # Mean Absolute Percentage Error (MAPE)

# Predictions and Model Names for Treated Water on Validation Set
treated_preds = [rf_pred_treated_validation, gb_pred_treated_validation, sarima_pred_treated_validation, ensemble_pred_treated_valida
treated_model_names = ['RF', 'GB', 'SARIMA', 'Ensemble']

for model_name, pred in zip(treated_model_names, treated_preds):
    calculate_metrics(y_validation_treated_actual, pred, model_name, 'Treated Water Validation')

# Predictions and Model Names for Energy Consumption on Validation Set
energy_preds = [rf_pred_energy_validation, gb_pred_energy_validation, sarima_pred_energy_validation, ensemble_pred_energy_validation]
energy_model_names = ['RF', 'GB', 'SARIMA', 'Ensemble']

for model_name, pred in zip(energy_model_names, energy_preds):
    calculate_metrics(y_validation_energy_actual, pred, model_name, 'Energy Consumption Validation')
```

```
=== RF for Treated Water Validation ===
RMSE: 0.11
MAE: 0.05
Error Rate: 0.41%

=== GB for Treated Water Validation ===
RMSE: 0.07
MAE: 0.03
Error Rate: 0.32%

=== SARIMA for Treated Water Validation ===
RMSE: 5.30
MAE: 4.99
Error Rate: 31.39%

=== Ensemble for Treated Water Validation ===
RMSE: 1.77
MAE: 1.66
Error Rate: 10.50%

=== RF for Energy Consumption Validation ===
RMSE: 4876.59
MAE: 3422.66
Error Rate: 9.53%

=== GB for Energy Consumption Validation ===
RMSE: 4332.31
MAE: 2619.35
Error Rate: 13.41%

=== SARIMA for Energy Consumption Validation ===
RMSE: 54726.09
MAE: 30672.51
Error Rate: 1585.03%

=== Ensemble for Energy Consumption Validation ===
RMSE: 19062.70
MAE: 11461.67
Error Rate: 531.42%
```

In [127]:

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'subsample': [0.8, 0.9, 1.0],
    'min_samples_split': [2, 3, 4]
}

# Create the model to be tuned
gb_base = GradientBoostingRegressor()

# Create the grid search object
grid_search = GridSearchCV(estimator = gb_base, param_grid = param_grid,
                           cv = 3, n_jobs = -1, verbose = 2)

# Fit the grid search to the data
grid_search.fit(X_train_actual, y_train_treated_actual)

# Get the best parameters
best_params = grid_search.best_params_
print("Best parameters:", best_params)

# Train and test the best model
best_gb_model_treated = GradientBoostingRegressor(**best_params)
best_gb_model_treated.fit(X_train_actual, y_train_treated_actual)
```

```
Fitting 3 folds for each of 243 candidates, totalling 729 fits
Best parameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 4, 'n_estimators': 50, 'subsample': 0.9}
```

Out[127]:

```
GradientBoostingRegressor(min_samples_split=4, n_estimators=50, subsample=0.9)
```

In [128]:

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'subsample': [0.8, 0.9, 1.0],
    'min_samples_split': [2, 3, 4]
}

# Create the model to be tuned
gb_base = GradientBoostingRegressor()

# Create the grid search object
grid_search = GridSearchCV(estimator = gb_base, param_grid = param_grid,
                           cv = 3, n_jobs = -1, verbose = 2)

# Fit the grid search to the data
grid_search.fit(X_train_actual, y_train_energy_actual)

# Get the best parameters
best_params = grid_search.best_params_
print("Best parameters:", best_params)

# Train and test the best model
best_gb_model_energy = GradientBoostingRegressor(**best_params)
best_gb_model_energy.fit(X_train_actual, y_train_energy_actual)
```

```
Fitting 3 folds for each of 243 candidates, totalling 729 fits
Best parameters: {'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 3, 'n_estimators': 100, 'subsample': 0.9}
```

Out[128]:

```
GradientBoostingRegressor(min_samples_split=3, subsample=0.9)
```

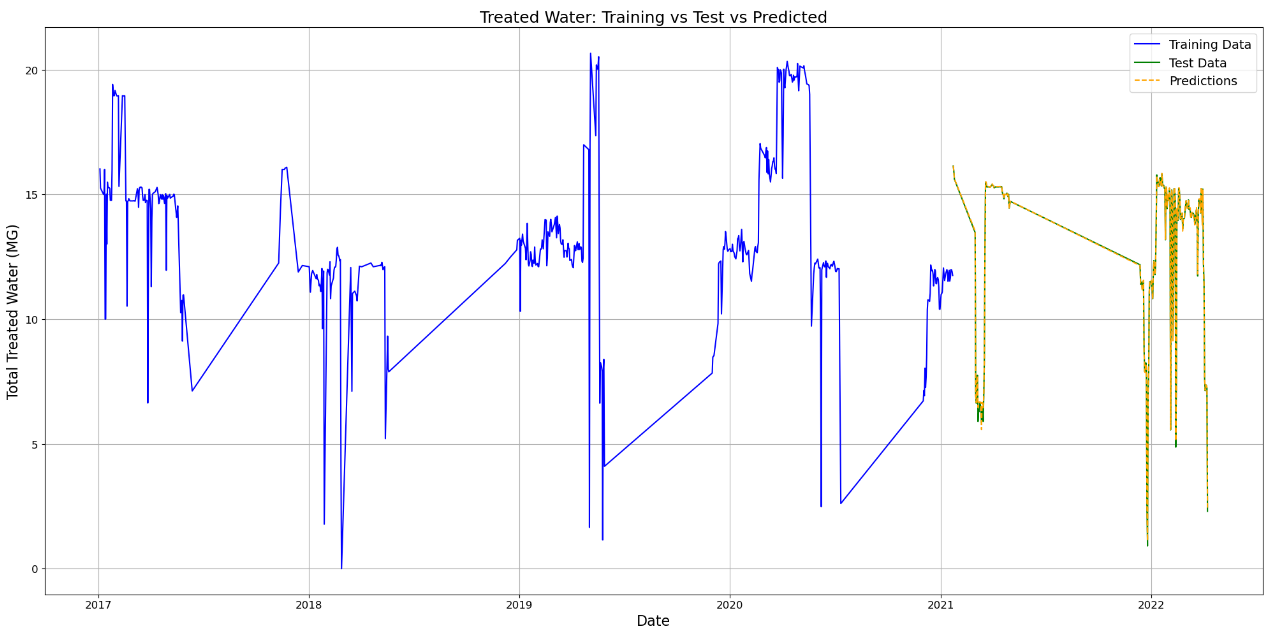
In [129]:

```
# Making predictions on the training set
y_pred_train_treated = best_gb_model.predict(X_train_actual)

# Calculate RMSE, MAE, and Error Rate for the training set
rmse_train_treated = sqrt(mean_squared_error(y_train_treated_actual, y_pred_train_treated))
mae_train_treated = mean_absolute_error(y_train_treated_actual, y_pred_train_treated)
error_train_treated = error(y_train_treated_actual, y_pred_train_treated)

# Making predictions on the test set (as you already did)
y_pred_tuned_treated = best_gb_model.predict(X_test_actual)

# Calculate RMSE, MAE, and Error Rate for the test set
rmse_test_treated = sqrt(mean_squared_error(y_test_treated_actual, y_pred_tuned_treated))
mae_test_treated = mean_absolute_error(y_test_treated_actual, y_pred_tuned_treated)
error_test_treated = error(y_test_treated_actual, y_pred_tuned_treated)

# Displaying the results
print("=== Hyperparameter-tuned Model Evaluation for Treated Water ===")
print("\n--- Training Set ---")
print(f"RMSE: {rmse_train_treated}")
print(f"MAE: {mae_train_treated}")
print(f"Error Rate: {error_train_treated}")

print("\n--- Test Set ---")
print(f"RMSE: {rmse_test_treated}")
print(f"MAE: {mae_test_treated}")
print(f"Error Rate: {error_test_treated}")

```

```
=== Hyperparameter-tuned Model Evaluation for Treated Water ===

--- Training Set ---
RMSE: 0.011638433832252987
MAE: 0.008573316842963906
Error Rate: 0.0

--- Test Set ---
RMSE: 0.10218589065663168
MAE: 0.05026823420561527
Error Rate: 0.01764705882352941
```

In [130]:

```
# Set the figure size
plt.figure(figsize=(20, 10))

# Plot Training Data
train_plot, = plt.plot(data['Date'][:len(y_train_treated_actual)], y_train_treated_actual, label='Training Data', color='blue')

# Plot Test Data
test_plot, = plt.plot(data['Date'][len(y_train_treated_actual):len(y_train_treated_actual) + len(y_test_treated_actual)], y_test_trea

# Overlay Predicted Data
predicted_plot, = plt.plot(data['Date'][len(y_train_treated_actual):len(y_train_treated_actual) + len(y_test_treated_actual)], y_pred

# Add titles, labels, and legend with increased font sizes
plt.title('Treated Water: Training vs Test vs Predicted', fontsize=18)
plt.xlabel('Date', fontsize=16)
plt.ylabel('Total Treated Water (MG)', fontsize=16)
plt.legend(handles=[train_plot, test_plot, predicted_plot], fontsize=14)
plt.grid(True)

# Increase the size of tick labels
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

# Show the plot
plt.tight_layout()
plt.show()

```



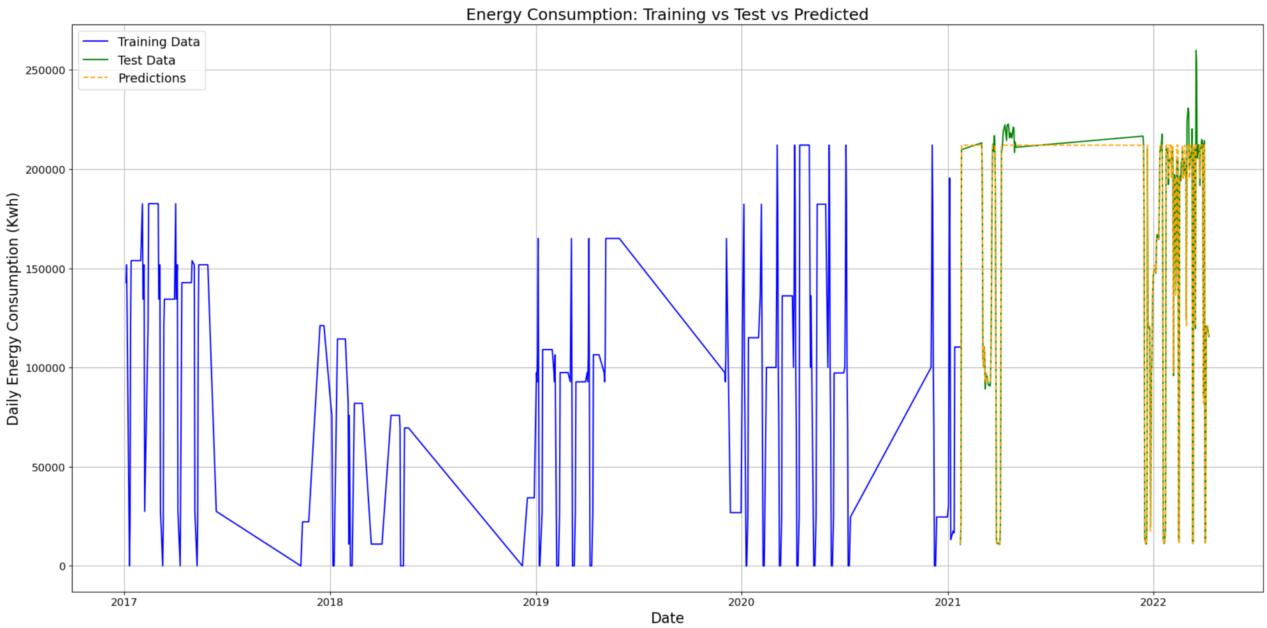
In [131]:

```
# Making predictions on the Training set for Energy Consumption
y_pred_train_energy = best_gb_model_energy.predict(X_train_actual)

# Compute the metrics for the training dataset
rmse_train_energy = sqrt(mean_squared_error(y_train_energy_actual, y_pred_train_energy))
mae_train_energy = mean_absolute_error(y_train_energy_actual, y_pred_train_energy)
error_rate_train_energy = error(y_train_energy_actual, y_pred_train_energy)

# Print out the metrics for training data
print("=== Training Data for Energy Consumption ===")
print(f"RMSE: {rmse_train_energy}")
print(f"MAE: {mae_train_energy}")
print(f"Error Rate: {error_rate_train_energy}")

# Making predictions on the test set for Energy Consumption
y_pred_test_energy = best_gb_model_energy.predict(X_test_actual)

# Compute the metrics for the test dataset
rmse_test_energy = sqrt(mean_squared_error(y_test_energy_actual, y_pred_test_energy))
mae_test_energy = mean_absolute_error(y_test_energy_actual, y_pred_test_energy)
error_rate_test_energy = error(y_test_energy_actual, y_pred_test_energy)

# Print out the metrics for test data
print("\n=== Test Data for Energy Consumption ===")
print(f"RMSE: {rmse_test_energy}")
print(f"MAE: {mae_test_energy}")
print(f"Error Rate: {error_rate_test_energy}")

```

```
=== Training Data for Energy Consumption ===
RMSE: 17.849240272718422
MAE: 5.251089131176049
Error Rate: 0.796078431372549

=== Test Data for Energy Consumption ===
RMSE: 6751.1928418417865
MAE: 3740.148612606303
Error Rate: 1.0
```

In [133]:

```
import matplotlib.pyplot as plt

# Set the figure size
plt.figure(figsize=(20, 10))

# Plot Training Data for Energy Consumption
train_plot, = plt.plot(data['Date'][:len(y_train_energy_actual)], y_train_energy_actual, label='Training Data', color='blue')

# Plot Test Data for Energy Consumption
test_plot, = plt.plot(data['Date'][len(y_train_energy_actual):len(y_train_energy_actual) + len(y_test_energy_actual)], y_test_energy_

# Overlay Predicted Data for Energy Consumption
predicted_plot, = plt.plot(data['Date'][len(y_train_energy_actual):len(y_train_energy_actual) + len(y_test_energy_actual)], y_pred_te

# Add titles, labels, and legend with increased font sizes
plt.title('Energy Consumption: Training vs Test vs Predicted', fontsize=18)
plt.xlabel('Date', fontsize=16)
plt.ylabel('Daily Energy Consumption (Kwh)', fontsize=16)
plt.legend(handles=[train_plot, test_plot, predicted_plot], fontsize=14)
plt.grid(True)

# Increase the size of tick labels
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

# Show the plot
plt.tight_layout()
plt.show()

```



In [134]:

```
import random

# Initialize lists to store predictions
treated_predictions = []
energy_predictions = []
```

In [135]:

```
# Number of loops needed to reach 1825 predictions
n_loops = 1825 // len(X_train_actual)  # Integer division to get full loops
remaining_rows = 1825 % len(X_train_actual)  # Remaining rows after full loops
```

In [136]:

```
# Loop through and concatenate the DataFrame with itself
X_future = pd.concat([X_train_actual] * n_loops, ignore_index=True)
```

In [137]:

```
# Add the remaining rows
if remaining_rows > 0:
    X_future = pd.concat([X_future, X_train_actual.iloc[:remaining_rows]], ignore_index=True)
```

In [138]:

```
# Make the predictions using the GB models
pred_treated = best_gb_model_treated.predict(X_future)
pred_energy = best_gb_model_energy.predict(X_future)

# Append the predictions to the lists
treated_predictions.extend(pred_treated)
energy_predictions.extend(pred_energy)

```

In [139]:

```
# Convert predictions to DataFrame
pred_df = pd.DataFrame({
    'Treated_Water_Predictions': treated_predictions,
    'Energy_Consumption_Predictions': energy_predictions
})
```

In [140]:

```
# Generate a date column
last_date = pd.to_datetime(data['Date'].iloc[-1])
future_dates = pd.date_range(start=last_date, periods=len(treated_predictions) + 1, freq='D')[1:]
```

In [141]:

```
# Add the 'Date' column and rearrange it to be the first column
pred_df['Date'] = future_dates
pred_df = pred_df[['Date', 'Treated_Water_Predictions', 'Energy_Consumption_Predictions']]
```

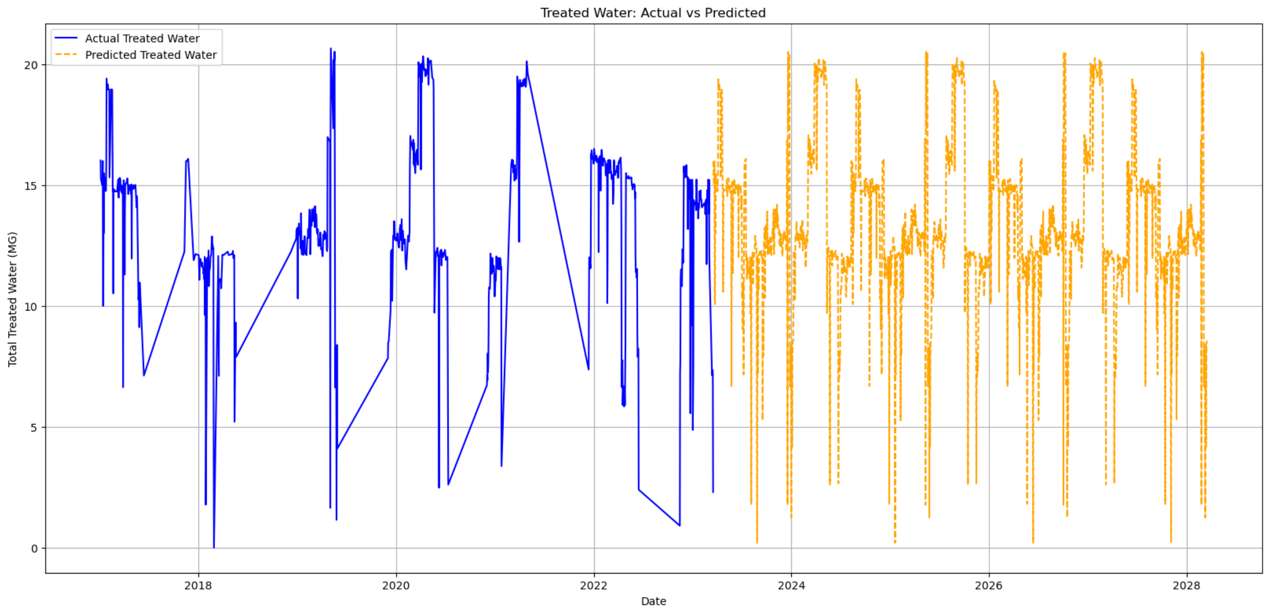
In [142]:

```
# Export to CSV
pred_df.to_csv('Future_Predictions.csv', index=False)

print("Predictions exported to Future_Predictions.csv")
```

```
Predictions exported to Future_Predictions.csv
```

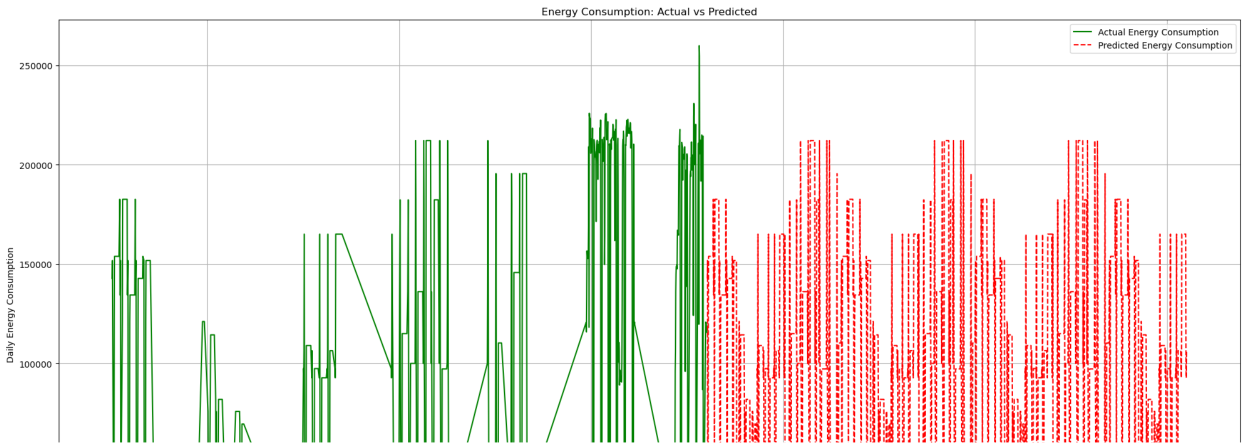
In [144]:

```
# Set the figure size
plt.figure(figsize=(20, 20))

# Plot Actual Treated Water
plt.subplot(2, 1, 1)

# Plot the training and test data (actual values)
plt.plot(data['Date'], data['TotalTreated water leaving plant,MG'], label='Actual Treated Water', color='blue')

# Overlay the test data predictions
plt.plot(pred_df['Date'], pred_df['Treated_Water_Predictions'], label='Predicted Treated Water', color='orange', linestyle='--')

plt.title('Treated Water: Actual vs Predicted')
plt.xlabel('Date')
plt.ylabel('Total Treated Water (MG)')
plt.legend()
plt.grid(True)

# Save the figure
plt.savefig("Predicted Treated water for Tampa Site.png", dpi=300, bbox_inches='tight')

plt.show()
```



In [145]:

```
# Set the figure size
plt.figure(figsize=(20, 10))

# Plot Training + Test Data for Actual Energy Consumption
plt.plot(data['Date'], data['Adjusted Daily Consumption, Kwh'], label='Actual Energy Consumption', color='green')

# Overlay Predicted Energy Consumption for Test Data
plt.plot(pred_df['Date'], pred_df['Energy_Consumption_Predictions'], label='Predicted Energy Consumption', color='red', linestyle='--

# Add titles, labels, and legend
plt.title('Energy Consumption: Actual vs Predicted')
plt.xlabel('Date')
plt.ylabel('Daily Energy Consumption')
plt.legend()
plt.grid(True)

# Show the plot
plt.tight_layout()
plt.show()

```



In [146]:

```
import pickle

# Saving the Gradient Boosting model for treated water in Tampa
with open('gb_model_treated_tampa.pkl', 'wb') as f:
    pickle.dump(best_gb_model_treated, f)

# Saving the Gradient Boosting model for energy consumption
with open('best_gb_model_energy_tampa.pkl', 'wb') as f:
    pickle.dump(best_gb_model_energy, f)

```

In [ ]:

```
# Loading the Gradient Boosting model for treated water
with open('gb_model_treated_tampa.pkl', 'rb') as f:
    loaded_gb_model_treated_tampa = pickle.load(f)

# Loading the Gradient Boosting model for energy consumption
with open('gb_model_energy_tampa.pkl', 'rb') as f:
    loaded_gb_model_energy_tampa = pickle.load(f)

```