Forecasting the Hospitality Employees

In [1]:
```
#Import Packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
%matplotlib inline
```

In [3]:
```
#Load Data
data= pd.read_csv(r'C:\Users\DD\Desktop\ML PROJECTS\time Series\HospitalityEmployees.csv')
```

In [4]:
```
data
```

Out[4]:

	Date	Employees
0	1/1/1990	1064.5
1	2/1/1990	1074.5
2	3/1/1990	1090.0
3	4/1/1990	1097.4
4	5/1/1990	1108.7
...	...	...
343	8/1/2018	2019.1
344	9/1/2018	1992.5
345	10/1/2018	1984.3
346	11/1/2018	1990.1
347	12/1/2018	2000.2

348 rows × 2 columns

#Data Preparation

In [5]:
```
data.shape
```

Out[5]:
```
(348, 2)
```

In [6]:
```
data.dtypes
```

Out[6]:
```
Date          object
Employees    float64
dtype: object
```

In [7]:
```
data.head()
```

Out[7]:

	Date	Employees
0	1/1/1990	1064.5
1	2/1/1990	1074.5
2	3/1/1990	1090.0
3	4/1/1990	1097.4
4	5/1/1990	1108.7

In [8]:
```
data.tail()
```

Out[8]:

	Date	Employees
343	8/1/2018	2019.1
344	9/1/2018	1992.5
345	10/1/2018	1984.3
346	11/1/2018	1990.1
347	12/1/2018	2000.2

In [9]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 348 entries, 0 to 347
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   Date       348 non-null    object 
 1   Employees  348 non-null    float64
dtypes: float64(1), object(1)
memory usage: 5.6+ KB
```

In [10]:
```
data.describe(include='all')
```

Out[10]:

	Date	Employees
count	348	348.000000
unique	348	NaN
top	1/1/1990	NaN
freq	1	NaN
mean	NaN	1452.506897
std	NaN	256.604914
min	NaN	1064.500000
25%	NaN	1238.050000
50%	NaN	1436.200000
75%	NaN	1586.300000
max	NaN	2022.100000

In [12]:
```
data.nunique()
```

Out[12]:
```
Date         348
Employees    338
dtype: int64
```

In [13]:
```
data.index.freq = 'MS'
data
```

Out[13]:

	Date	Employees
0	1/1/1990	1064.5
1	2/1/1990	1074.5
2	3/1/1990	1090.0
3	4/1/1990	1097.4
4	5/1/1990	1108.7
...	...	...
343	8/1/2018	2019.1
344	9/1/2018	1992.5
345	10/1/2018	1984.3
346	11/1/2018	1990.1
347	12/1/2018	2000.2

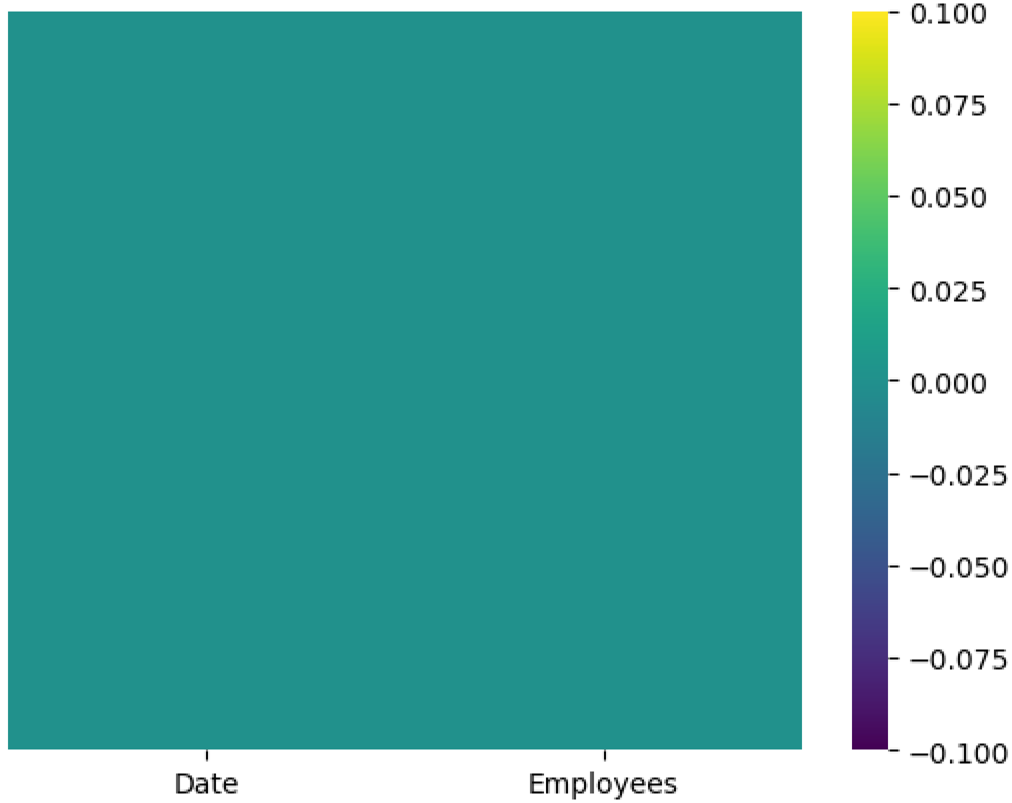
348 rows × 2 columns

In [11]:
```
data.isnull().sum()
```

Out[11]:
```
Date         0
Employees    0
dtype: int64
```

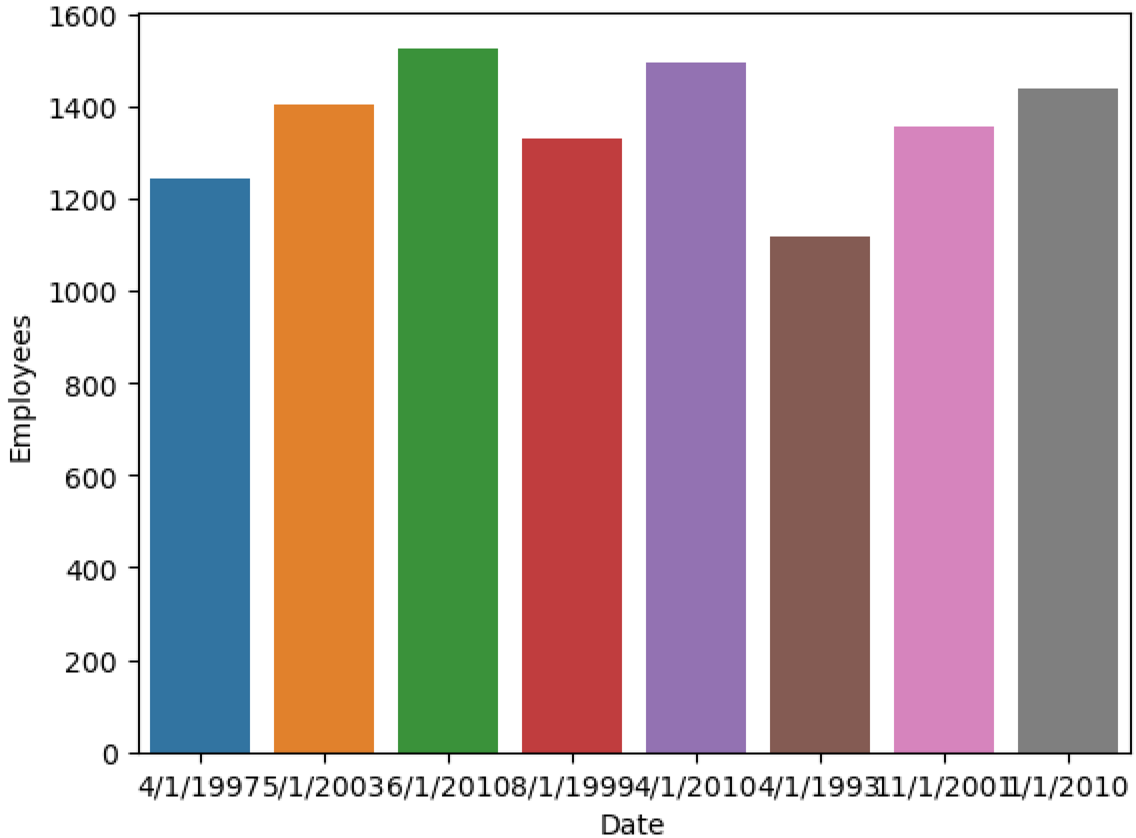
In [50]:
```
sns.heatmap(data.isnull(),yticklabels=False,cmap="viridis")
```

Out[50]: <Axes: >

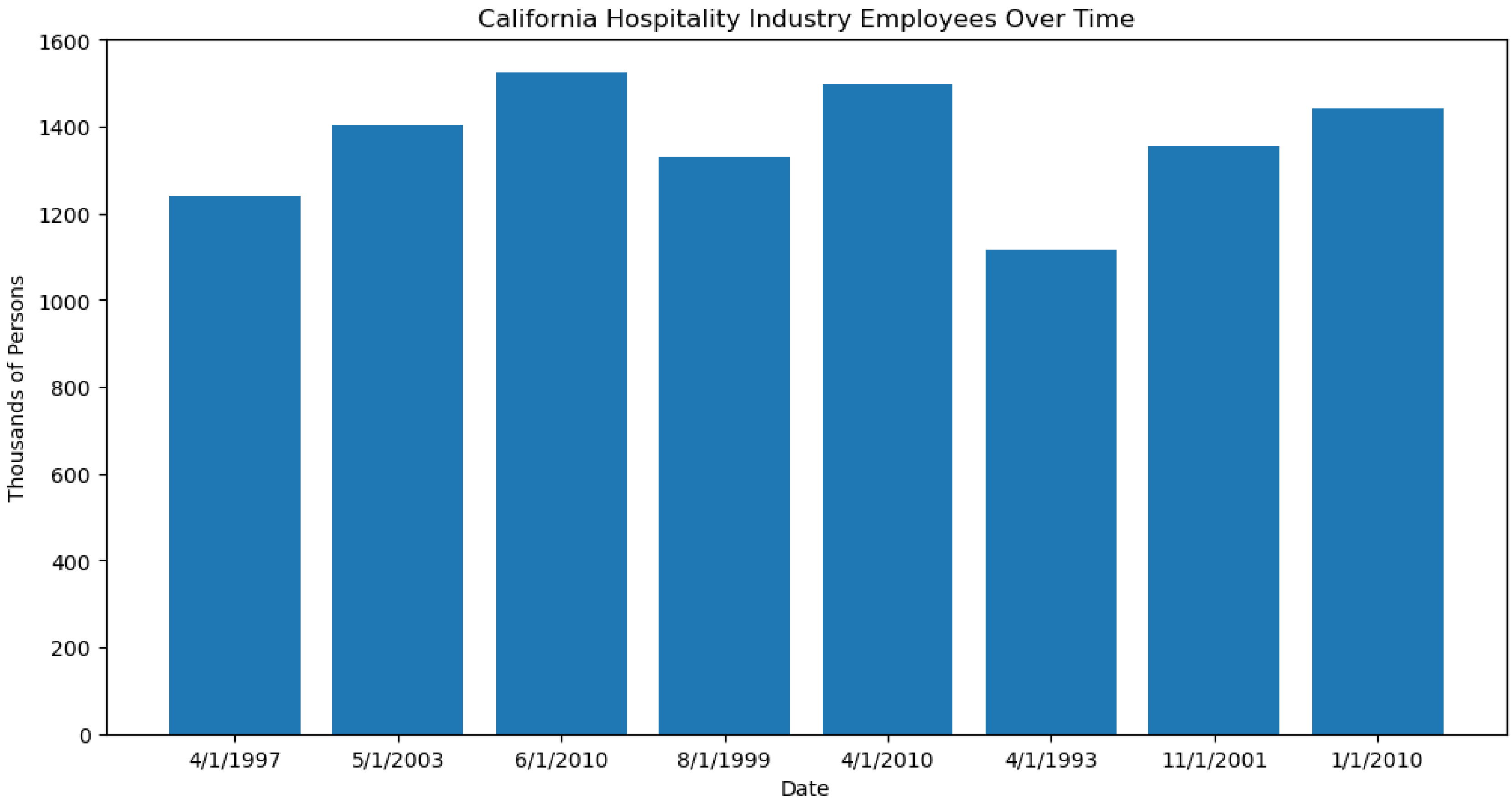


In [14]:
```
s=data.sample(8)
sns.barplot(x='Date',y='Employees',data=s)
```

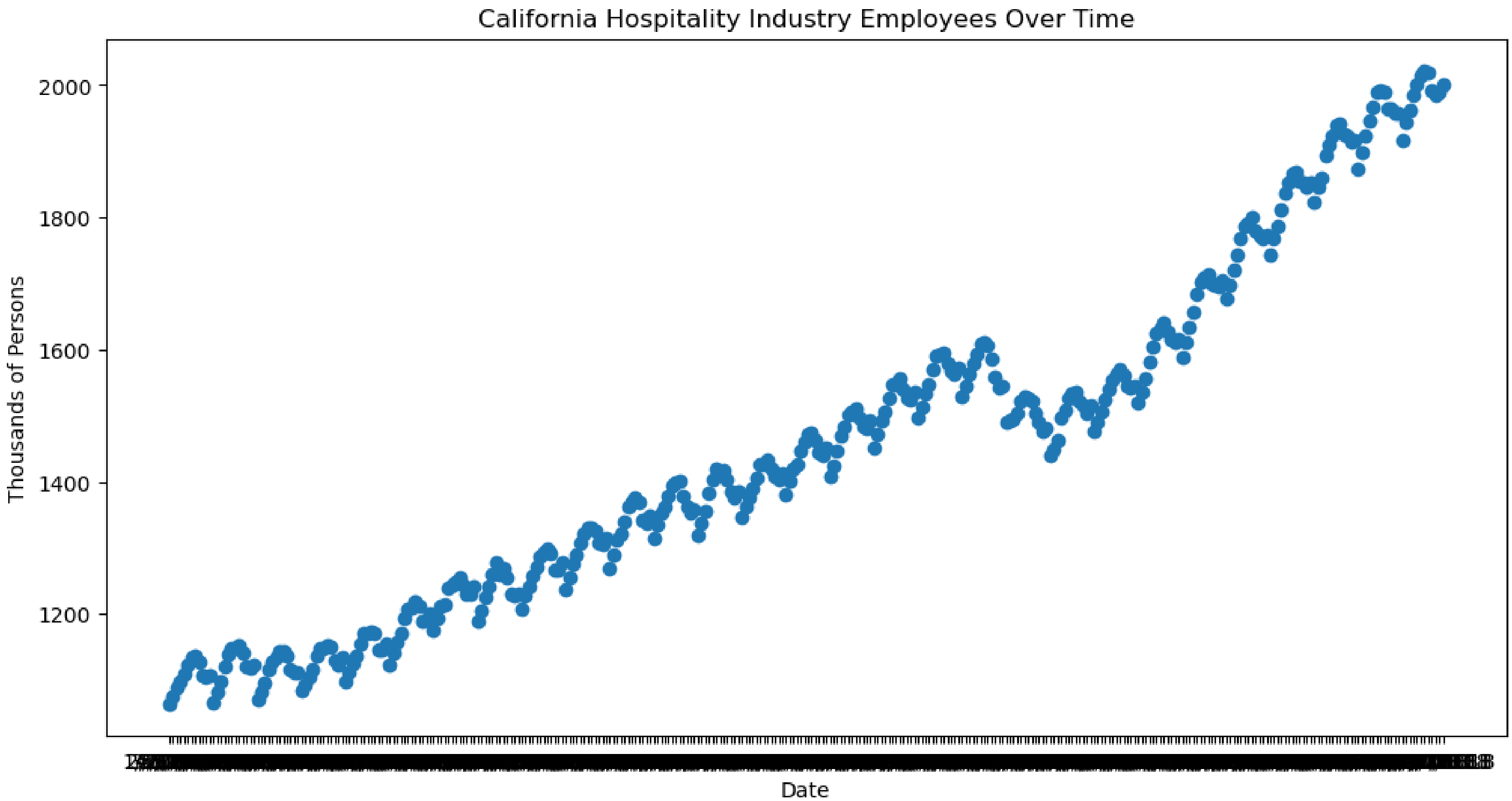
Out[14]: <Axes: xlabel='Date', ylabel='Employees'>



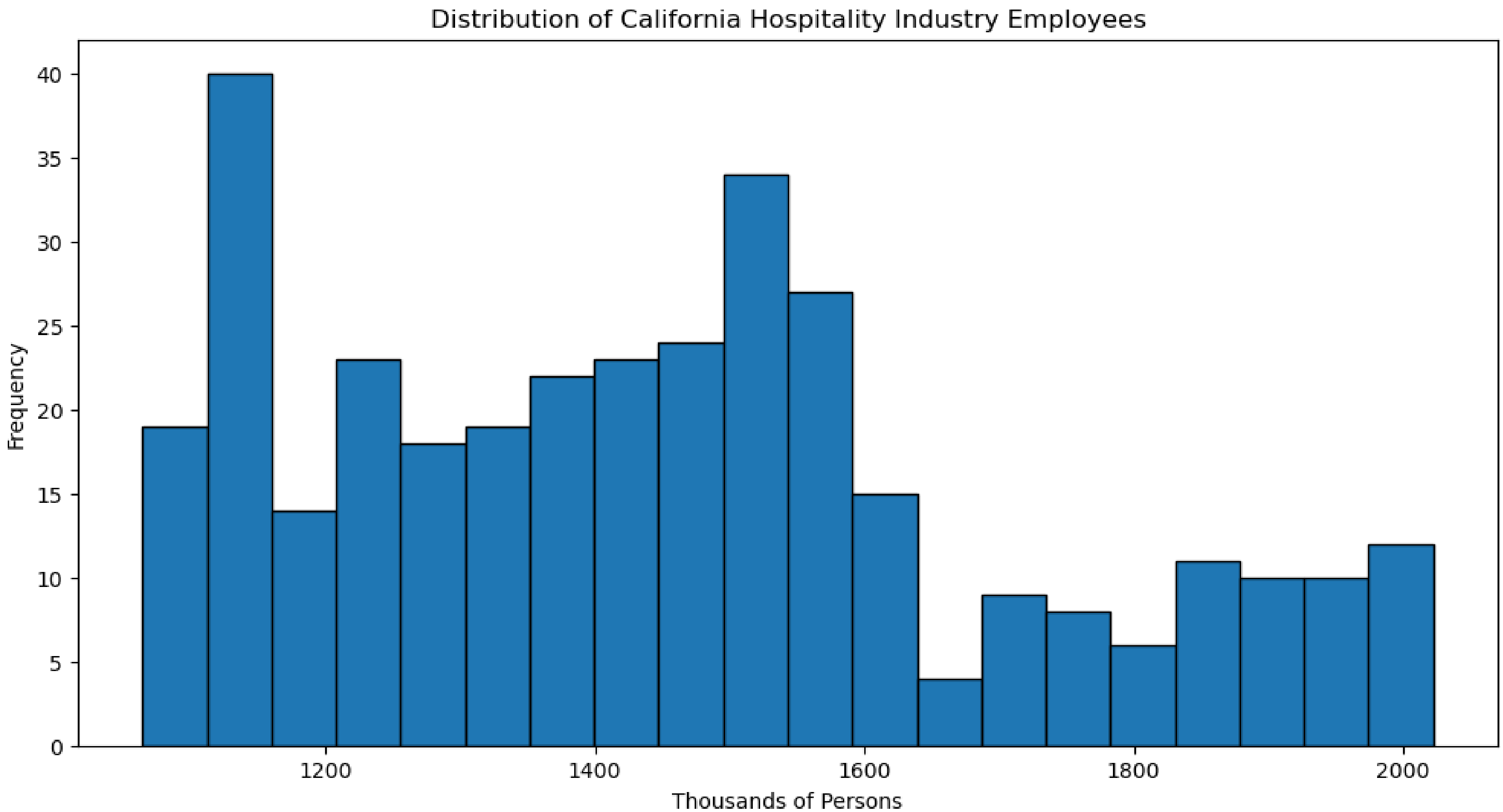
In [23]:
```
plt.figure(figsize=(12, 6))
plt.bar(s['Date'],s['Employees'])
plt.title('California Hospitality Industry Employees Over Time')
plt.xlabel('Date')
plt.ylabel('Thousands of Persons')
plt.show()
```



In [25]:
```
plt.figure(figsize=(12, 6))
plt.scatter(data['Date'], data['Employees'], marker='o')
plt.title('California Hospitality Industry Employees Over Time')
plt.xlabel('Date')
plt.ylabel('Thousands of Persons')
plt.show()
```

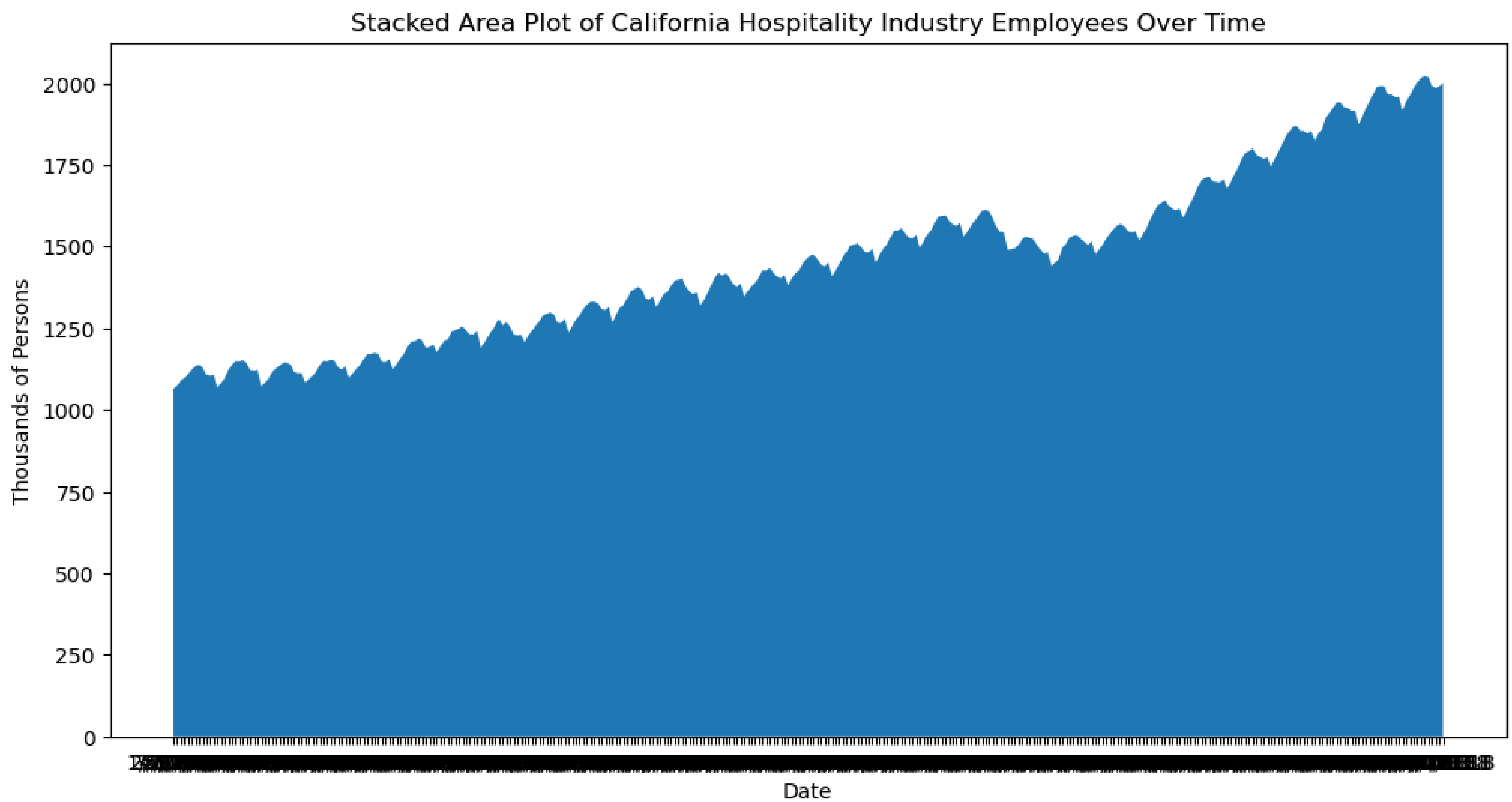


In [17]:
```
plt.figure(figsize=(12, 6))
plt.hist(data['Employees'], bins=20, edgecolor='black')
plt.title('Distribution of California Hospitality Industry Employees')
plt.xlabel('Thousands of Persons')
plt.ylabel('Frequency')
plt.show()
```



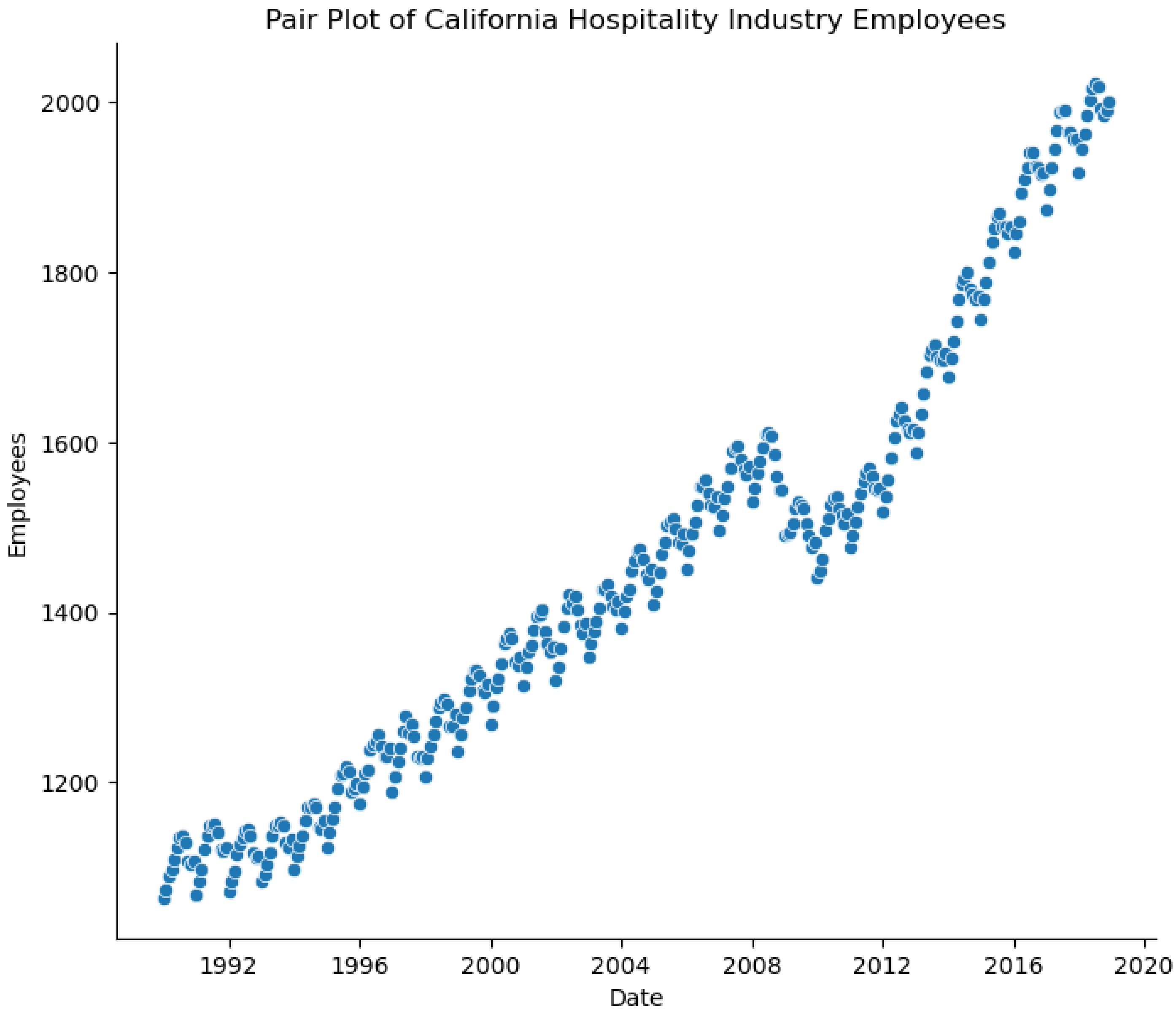
In [26]:
```
plt.figure(figsize=(12, 6))
plt.boxplot(s['Employees'])
plt.title('Box Plot of California Hospitality Industry Employees')
plt.ylabel('Thousands of Persons')
plt.show()
```

In [19]:
```
plt.figure(figsize=(12, 6))
plt.stackplot(data['Date'], data['Employees'])
plt.title('Stacked Area Plot of California Hospitality Industry Employees Over Time')
plt.xlabel('Date')
plt.ylabel('Thousands of Persons')
plt.show()
```

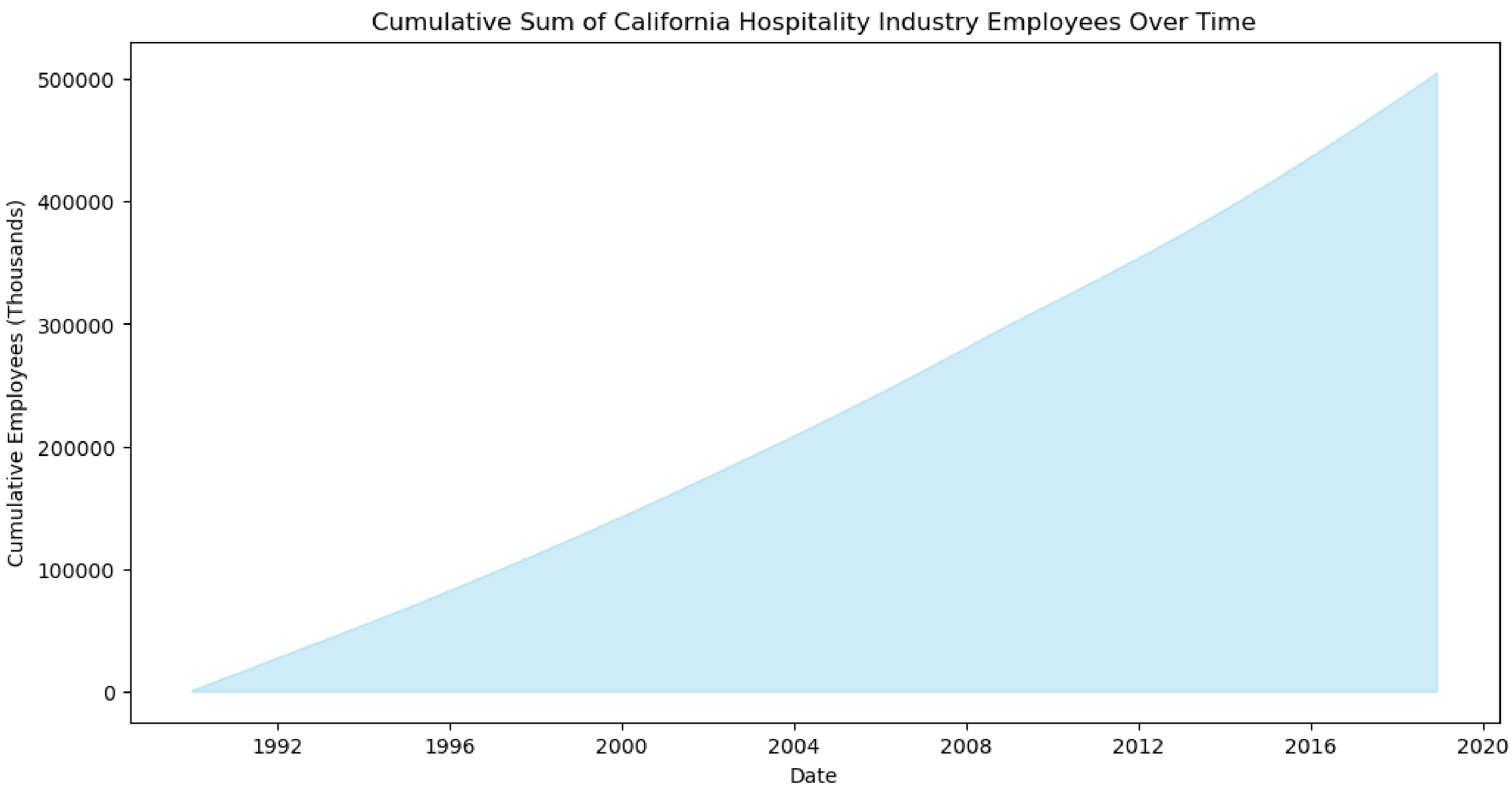


In [31]:
```
sns.pairplot(data, x_vars=['Date'], y_vars=['Employees'], height=6, aspect=1.2, kind='scatter')
plt.title('Pair Plot of California Hospitality Industry Employees')
plt.show()
```

```
C:\Users\DD\AppData\Local\anaconda3\Lib\site-packages\seaborn\axisgrid.py:118: UserWarning: The figure layou
t has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```


Pair Plot of California Hospitality Industry Employees
Employees
2000
1800
1600
1400
1200
1992
1996
2000
2004
2008
2012
2016
2020
Date

In [32]:
```
plt.figure(figsize=(12, 6))
plt.fill_between(data['Date'], data['Employees'].cumsum(), color='skyblue', alpha=0.4)
plt.title('Cumulative Sum of California Hospitality Industry Employees Over Time')
plt.xlabel('Date')
plt.ylabel('Cumulative Employees (Thousands)')
plt.show()
```



Indexing with Date

This is particularly helpful for calculating statistics or summarizing data over specific time periods

In [33]:
```
data = data.groupby('Date')['Employees'].sum().reset_index()
```

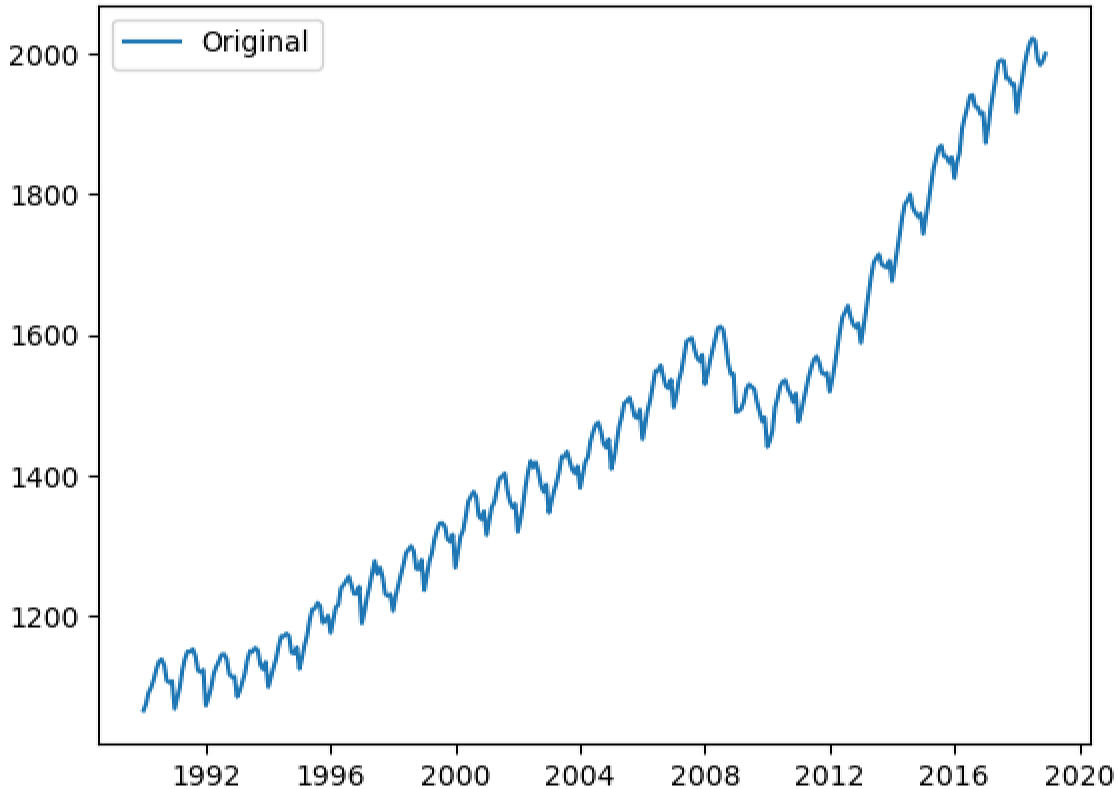
In [34]:
```
data['Date'] = pd.to_datetime(data['Date'])
data.set_index('Date', inplace=True)
data.index
```

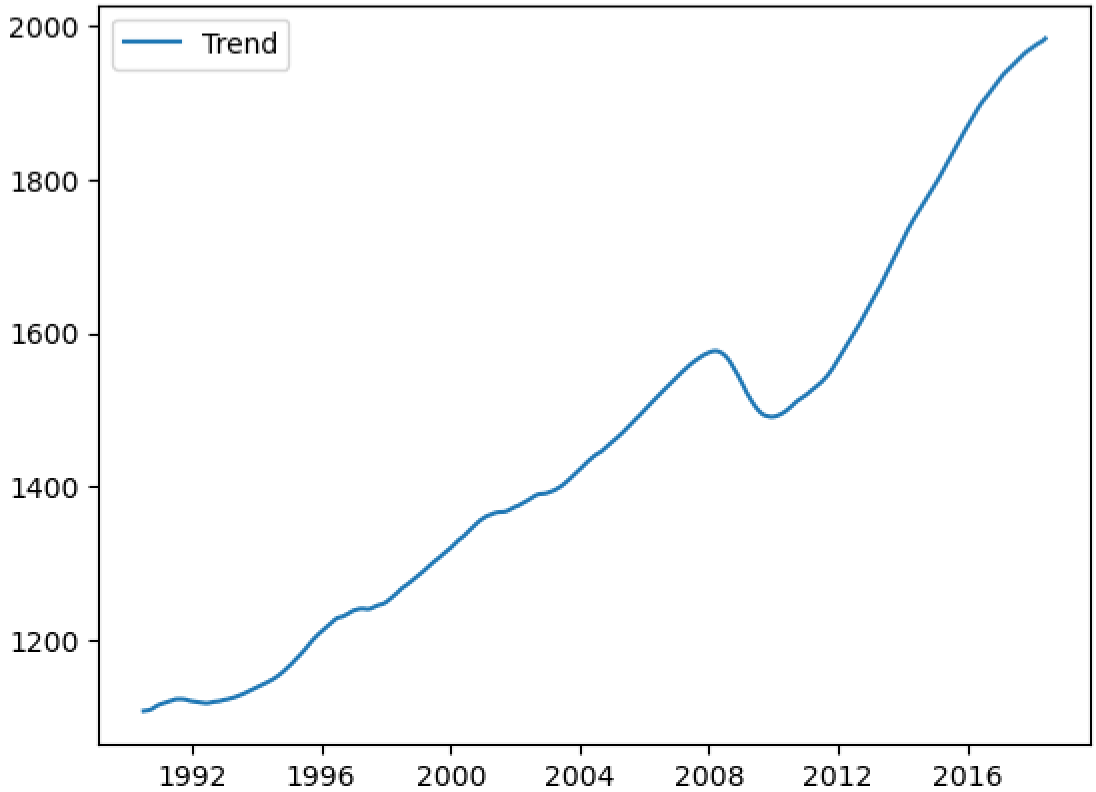
Out[34]:
```
DatetimeIndex(['1990-01-01', '1990-02-01', '1990-03-01', '1990-04-01',
               '1990-05-01', '1990-06-01', '1990-07-01', '1990-08-01',
               '1990-09-01', '1990-10-01',
               ...
               '2018-03-01', '2018-04-01', '2018-05-01', '2018-06-01',
               '2018-07-01', '2018-08-01', '2018-09-01', '2018-10-01',
               '2018-11-01', '2018-12-01'],
              dtype='datetime64[ns]', name='Date', length=348, freq=None)
```

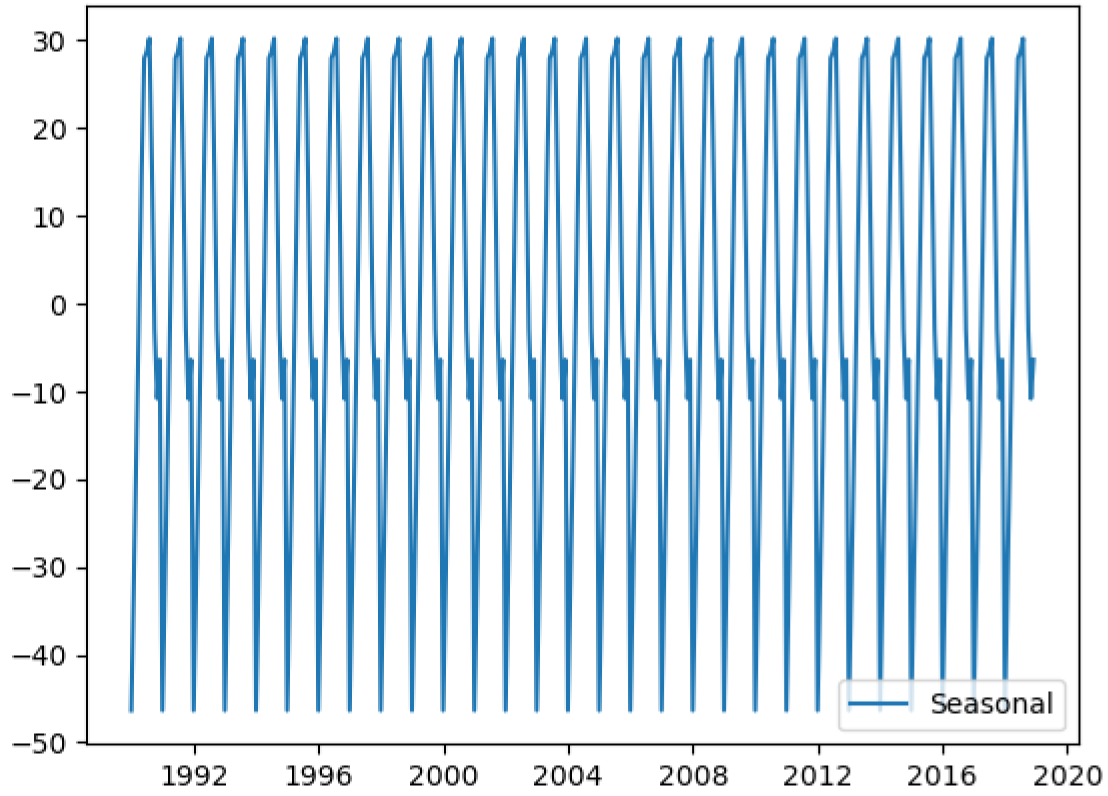
Sampling

In [35]:
```
y = data['Employees'].resample('MS').mean()
```

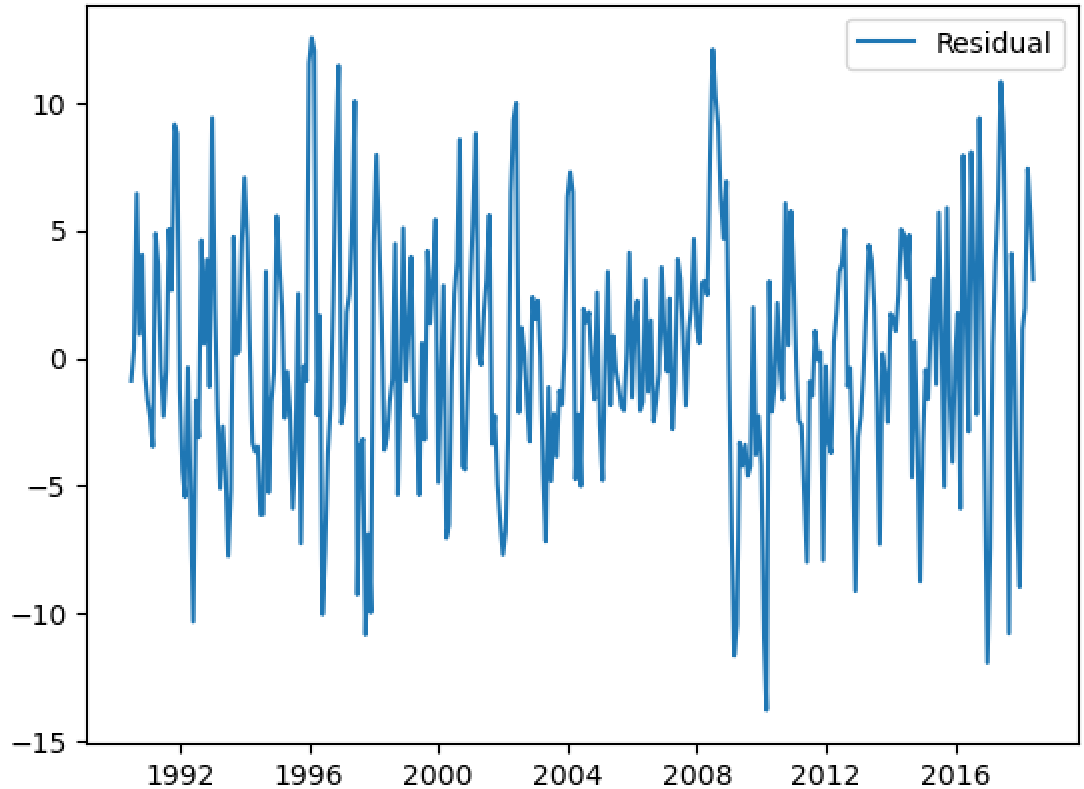
In [36]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(y)

plt.plot(y, label = 'Original')
plt.legend(loc = 'best')

trend = decomposition.trend
plt.show()
plt.plot(trend, label = 'Trend')
plt.legend(loc = 'best')

seasonal = decomposition.seasonal
plt.show()
plt.plot(seasonal, label = 'Seasonal')
plt.legend(loc = 'best')

residual = decomposition.resid
plt.show()
plt.plot(residual, label = 'Residual')
plt.legend(loc='best')
```


Original
2000
1800
1600
1400
1200
1992
1996
2000
2004
2008
2012
2016
2020


Trend
2000
1800
1600
1400
1200
1992
1996
2000
2004
2008
2012
2016



Out[36]: <matplotlib.legend.Legend at 0x2038763b3d0>



Checking Stationarity

In [37]:
```
from statsmodels.tsa.stattools import adfuller
```

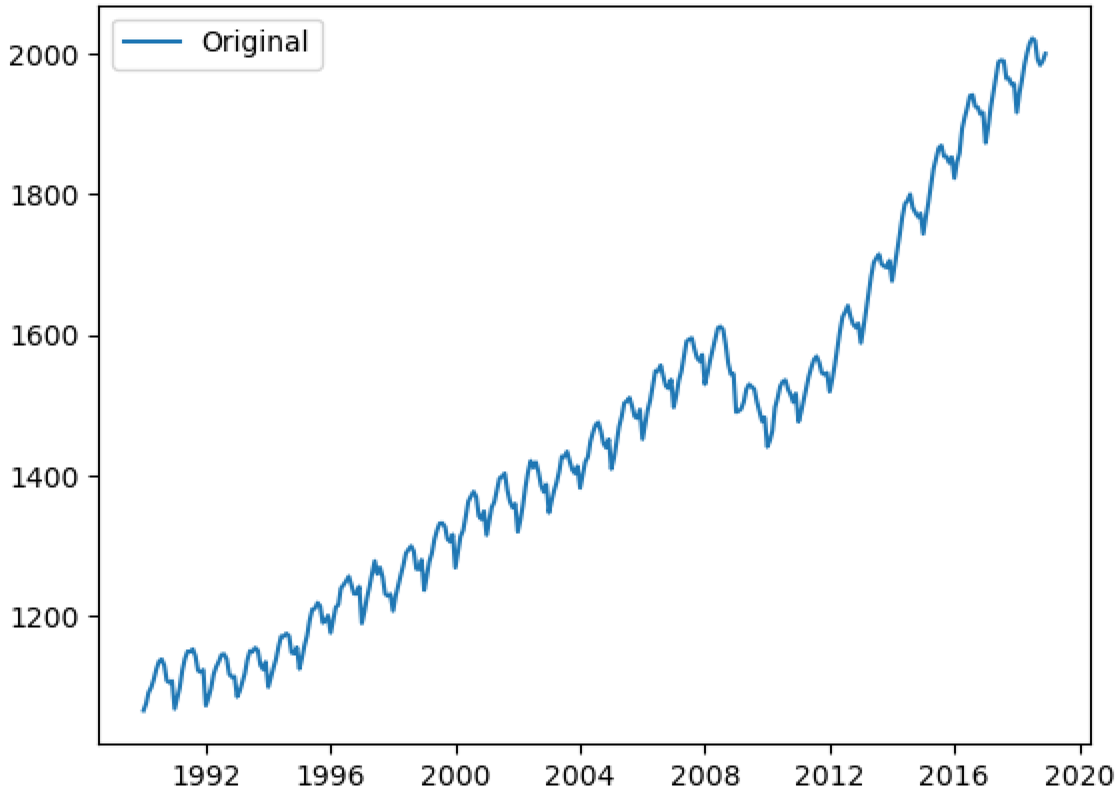
In [38]:
```
from pandas import Series
from statsmodels.tsa.stattools import adfuller
result = adfuller(y)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: 0.901284
p-value: 0.993107
Critical Values:
    1%: -3.450
    5%: -2.870
    10%: -2.571
```

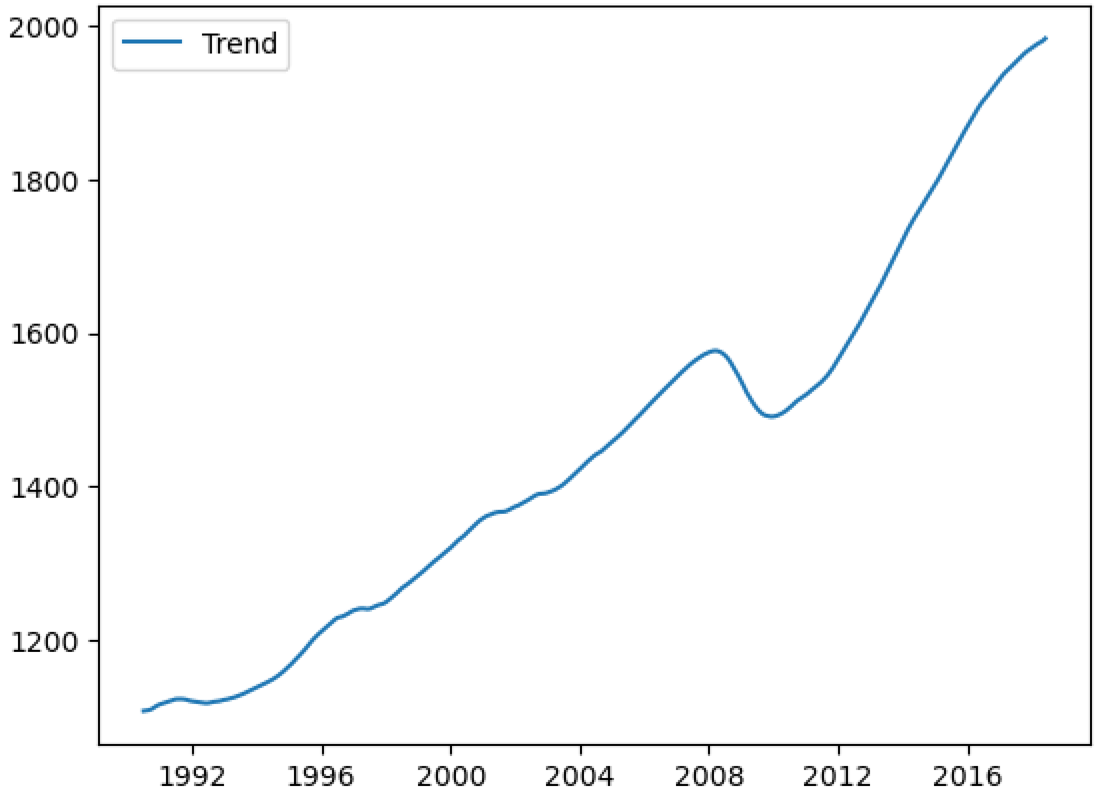
Decomposing

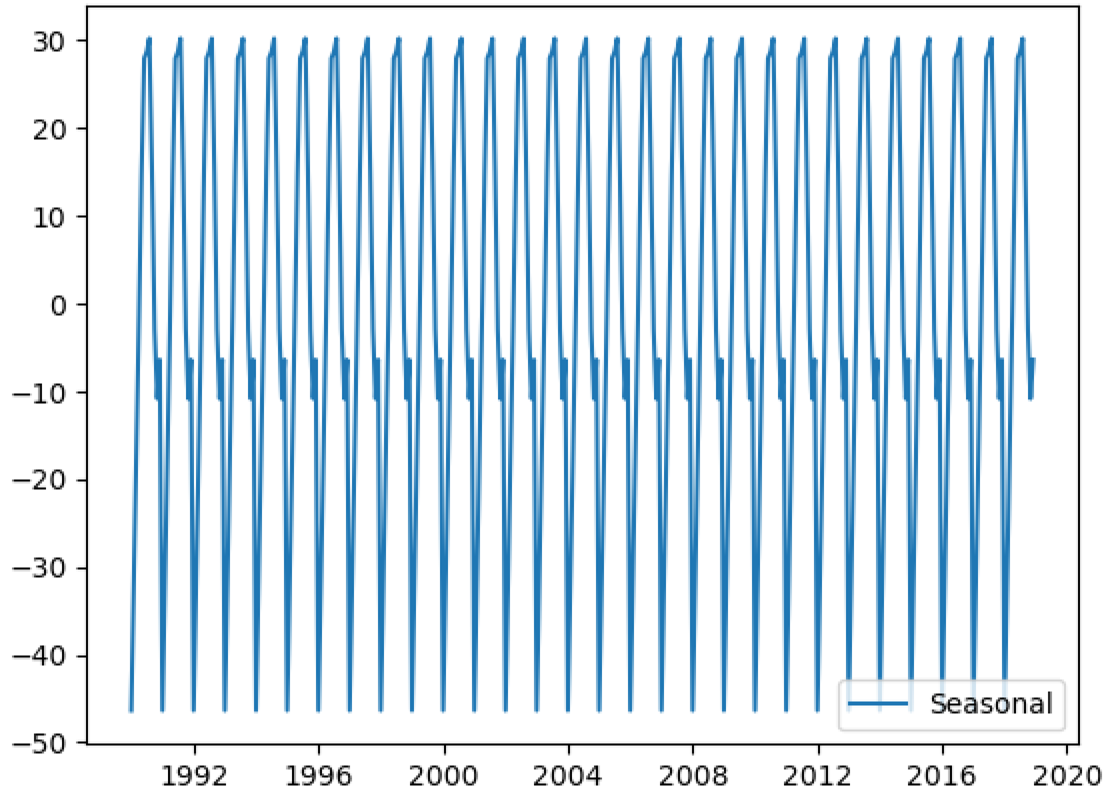
Decomposing the time series into three distinct components: trend, seasonality, and noise

In [39]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(y)

plt.plot(y, label = 'Original')
plt.legend(loc = 'best')

trend = decomposition.trend
plt.show()
plt.plot(trend, label = 'Trend')
plt.legend(loc = 'best')

seasonal = decomposition.seasonal
plt.show()
plt.plot(seasonal, label = 'Seasonal')
plt.legend(loc = 'best')

residual = decomposition.resid
plt.show()
plt.plot(residual, label = 'Residual')
plt.legend(loc='best')
```

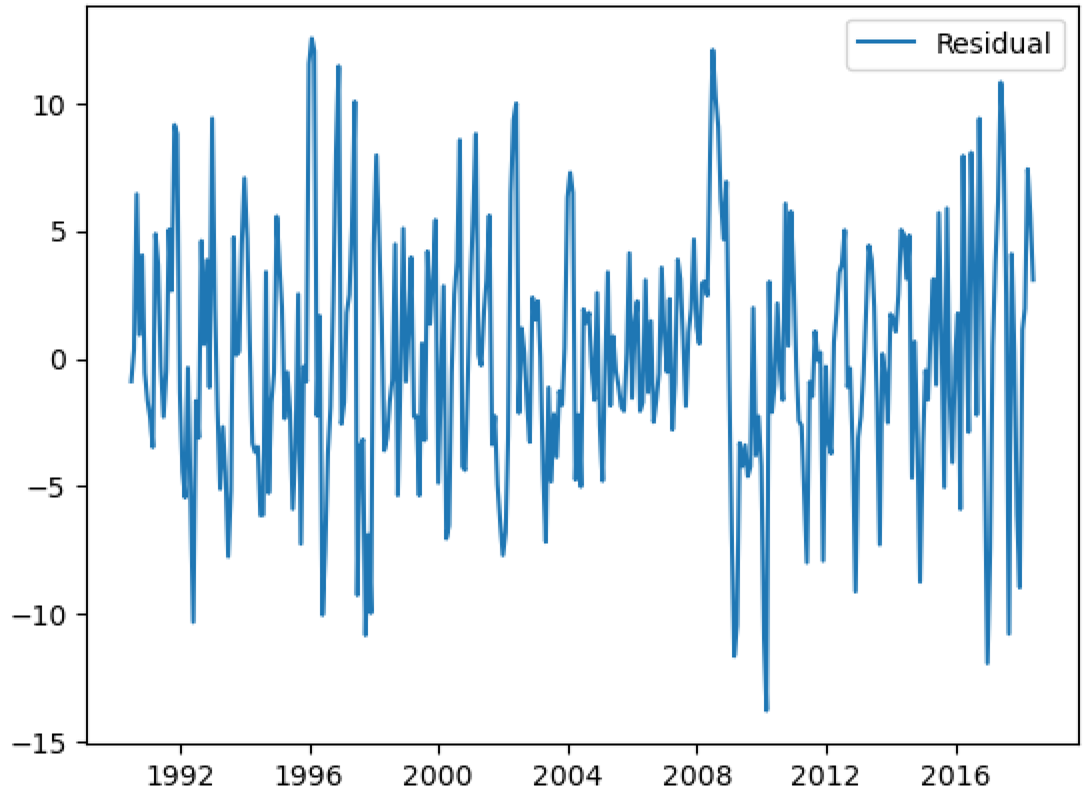


Original
2000
1800
1600
1400
1200
1992
1996
2000
2004
2008
2012
2016
2020


Trend
2000
1800
1600
1400
1200
1992
1996
2000
2004
2008
2012
2016



Out[39]: <matplotlib.legend.Legend at 0x203887f5cd0>



Time Series Forcasting using ARIMA

#We will use ARIMA for forecasting our time series. ARIMA is also denoted as ARIMA(p,d,q) where p,d,q accounts for seasonality, trend and noise in the time series data

In [40]:
```
import itertools
p = d = q = range(0, 2)
pdq = list(itertools.product(p, d, q))
seasonal_pdq = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))]
print('Examples of parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[1]))
print('SARIMAX: {} x {}'.format(pdq[1], seasonal_pdq[2]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[3]))
print('SARIMAX: {} x {}'.format(pdq[2], seasonal_pdq[4]))
```

```
Examples of parameter combinations for Seasonal ARIMA...
SARIMAX: (0, 0, 1) x (0, 0, 1, 12)
SARIMAX: (0, 0, 1) x (0, 1, 0, 12)
SARIMAX: (0, 1, 0) x (0, 1, 1, 12)
SARIMAX: (0, 1, 0) x (1, 0, 0, 12)
```

Parameter Selection

We use “grid search” to find the optimal set of parameters that yields the best performance for our model

In [41]:
```
from pylab import rcParams
for param in pdq:
    for param_seasonal in seasonal_pdq:
        try:
            mod = sm.tsa.statespace.SARIMAX(y, order=param,
seasonal_order=param_seasonal,
enforce_stationarity=False,
enforce_invertibility=False)
            results = mod.fit()
            print('ARIMA{}x{}12 - AIC:{}'.format(param, param_seasonal, results.aic))
        except:
            continue
```

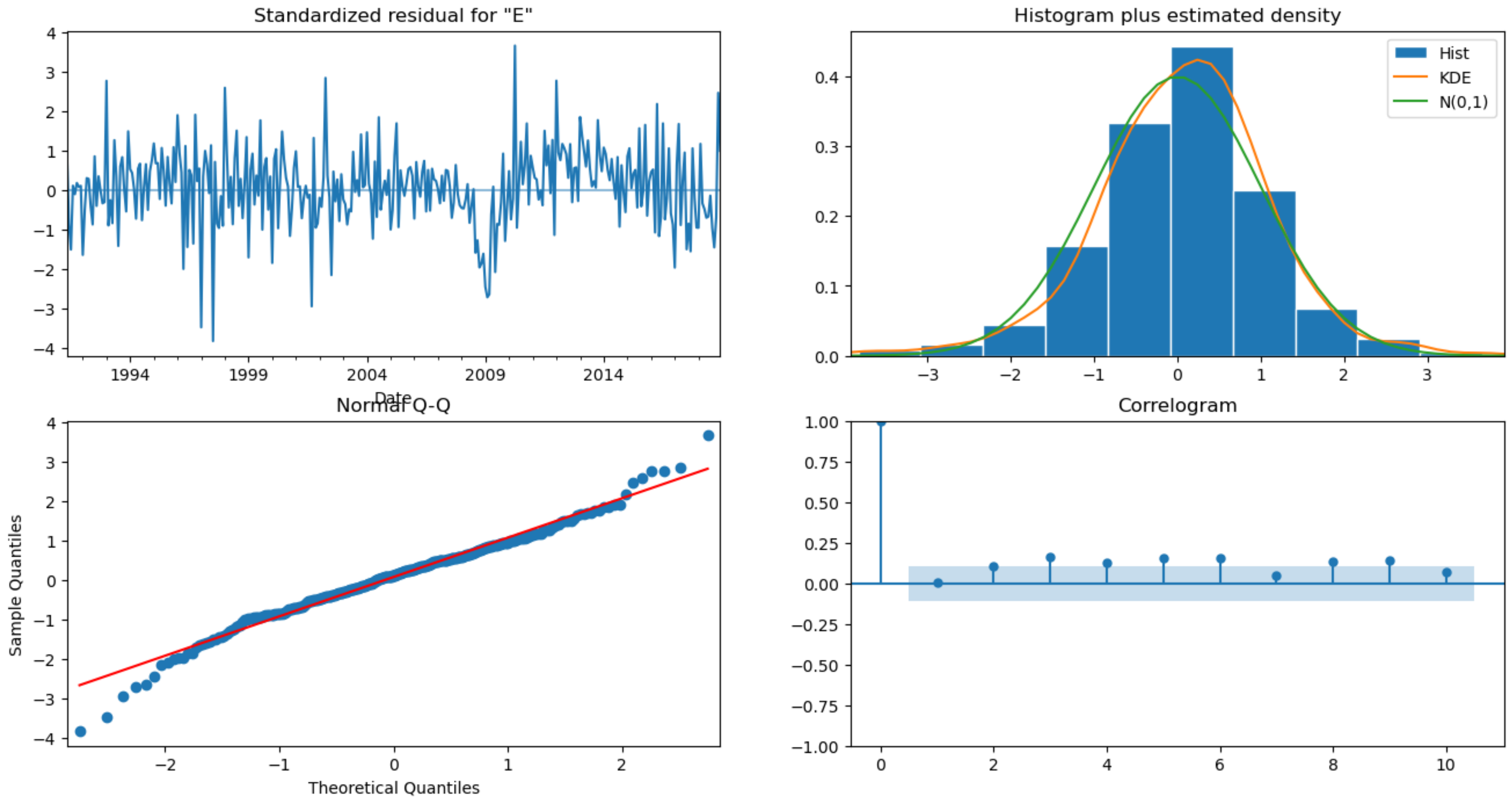
Fitting the ARIMA model

We might use the ARIMA Order (1, 1, 2) combined with the Seasonal Order (1, 0, [1], 12) as we have the lowest AIC value considering those orders

In [51]:
```
import statsmodels.api as sm
mod = sm.tsa.statespace.SARIMAX(y,
                                order=(1, 1, 2),
                                seasonal_order=(1, 0, 1, 12),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
results = mod.fit()
print(results.summary().tables[1])
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.5069      0.449     -1.129      0.259      -1.387       0.373
ma.L1          3.1335     10.333      0.303      0.762     -17.118      23.385
ma.L2          0.5887      3.993      0.147      0.883      -7.237       8.415
ar.S.L12       0.9996      0.005    206.218      0.000       0.990       1.009
ma.S.L12      -0.7189      0.042    -17.013      0.000      -0.802      -0.636
sigma2         3.6716     24.122      0.152      0.879     -43.607      50.950
==============================================================================

C:\Users\DD\AppData\Local\anaconda3\Lib\site-packages\statsmodels\base\model.py:607: ConvergenceWarning: Max
imum Likelihood optimization failed to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

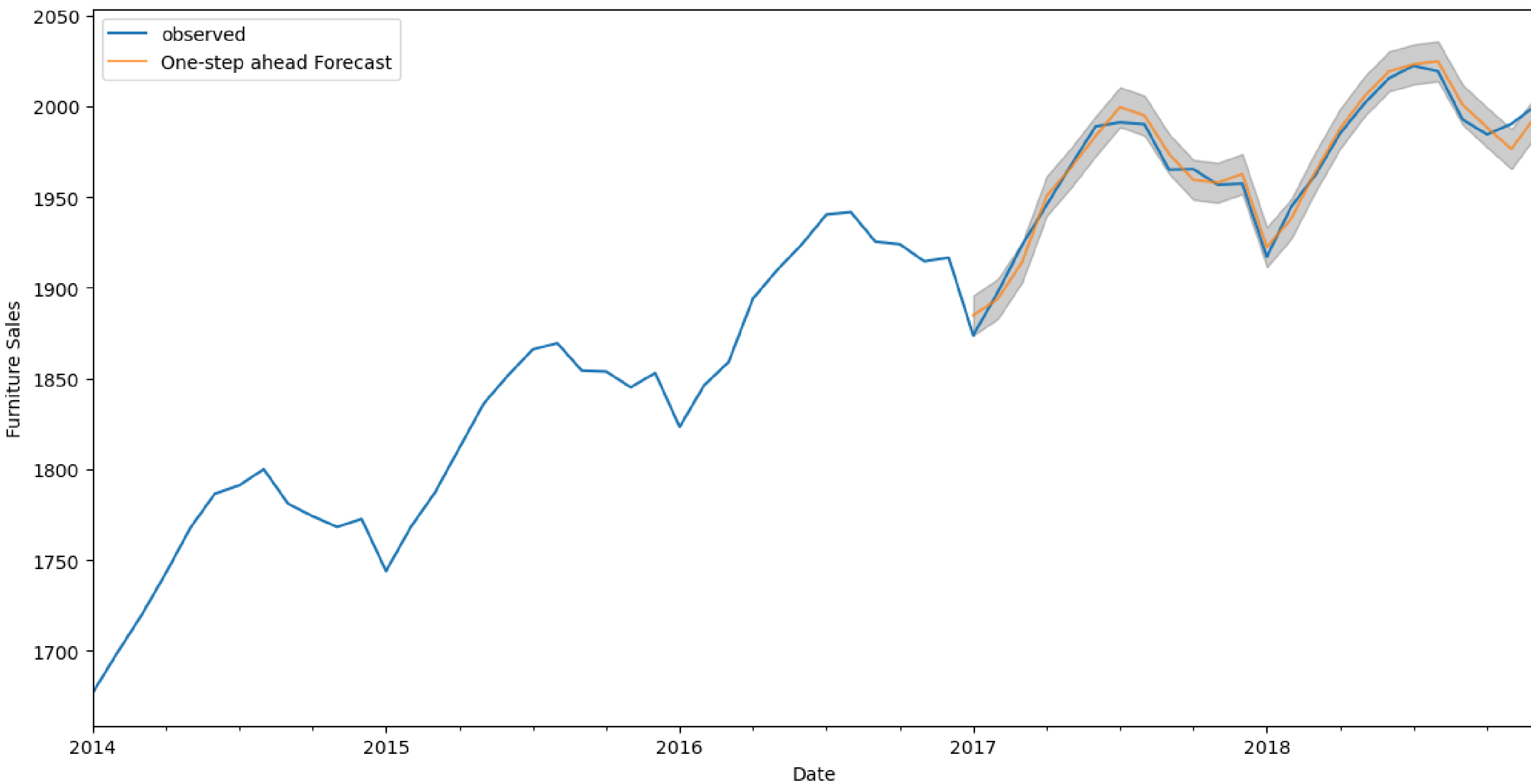
In [52]:
```
results.plot_diagnostics(figsize=(16, 8))
plt.show()
```



Validating Forecasts

```
We compare predicted sales to real sales of the time series to understand the accuracy of our forecasts
```

In [53]:
```
#set forecasts to start at 2017-01-01 to the end of the data to forecast
pred = results.get_prediction(start=pd.to_datetime('2017-01-01'), dynamic=False)
pred_ci = pred.conf_int()
ax = y['2014':].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead Forecast', alpha=.7, figsize=(14, 7))
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.2)
ax.set_xlabel('Date')
ax.set_ylabel('Thousands of Persons')
plt.legend()
plt.show()
```



Calculating MSE and RMSE

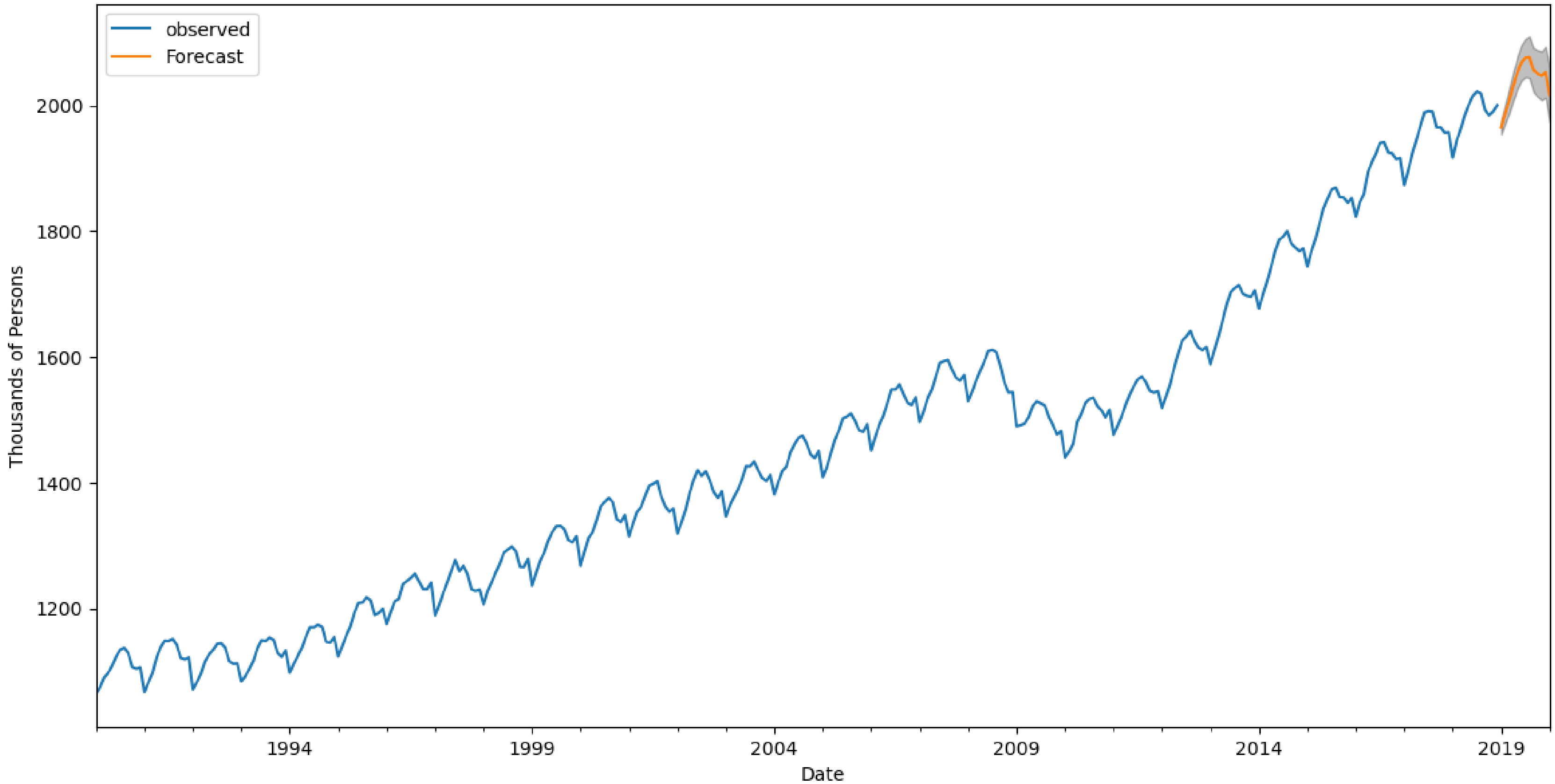
In [54]:
```
y_data = pred.predicted_mean
y_truth = y['2017-01-01':]
mse = ((y_data - y_truth) ** 2).mean()
print('The Mean Squared Error of our forecasts is {}'.format(round(mse, 2)))

print('The Root Mean Squared Error of our forecasts is {}'.format(round(np.sqrt(mse), 2)))
```

```
The Mean Squared Error of our forecasts is 40.14
The Root Mean Squared Error of our forecasts is 6.34
```

Visualizing the Forecast

In [58]:
```
pred_uc = results.get_forecast(steps=13)
pred_ci = pred_uc.conf_int()
ax = y.plot(label='observed', figsize=(14, 7))
pred_uc.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='k', alpha=.25)
ax.set_xlabel('Date')
ax.set_ylabel('Thousands of Persons')
print(pred_ci)
plt.legend()
plt.show()
```

```
            lower Employees  upper Employees
2019-01-01      1953.968474      1975.996741
2019-02-01      1972.220508      2003.920391
2019-03-01      1987.319019      2027.027686
2019-04-01      2008.480242      2054.544035
2019-05-01      2025.373608      2077.146884
2019-06-01      2039.424153      2096.276643
2019-07-01      2044.708184      2106.250152
2019-08-01      2043.526600      2109.411798
2019-09-01      2020.918310      2090.884002
2019-10-01      2013.989476      2087.807386
2019-11-01      2008.298797      2085.779112
2019-12-01      2012.329460      2093.305989
2020-01-01      1973.819716      2060.078685
```



Conclusion

We observe that the Average of employees produce seasonal pattern and the Average of employees increases linearly over time.