Independent Statistics & Analysis
U.S. Energy Information Administration

API Technical Documentation

API v2.1.0

November 2022

Independent Statistics & Analysis
www.eia.gov

U.S. Department of Energy
Washington, DC 20585

This report was prepared by the U.S. Energy Information Administration (EIA), the statistical and analytical agency within the U.S. Department of Energy. By law, EIA's data, analyses, and forecasts are independent of approval by any other officer or employee of the U.S. Government. The views in this report should not be construed as representing those of the U.S. Department of Energy or other federal agencies.

APIv2 Documentation Changelog

November 2022, v2.1.0: Clementia
Feature patch to add backward compatibility and migration features for APIv1.

- Added automatic translation for APIv1 series IDs with the v2 /SeriesID path
- Introduced in-return warning messages

November 2022, v2.0.4: Performance Tuning

- Optimized the back-end route processor to improve performance, especially while under load.

September 2022, v2.0.3: Minor bux fixes.

- Eliminated a bug where use of the **`length`** parameter could returm more than the 5,000 row maximum.

May 2022, v2.0.2: Public Release

- Added Support for XML output with the **`out=xml`** parameter.

February 2022, v2.0.1: Community Release

- Released first spec of APIv2; no changes.

Introduction

EIA launched its original API in 2012. Since then, internet protocols and standards have evolved. To keep pace with these best practices and provide new, modern features, we have released a second version of our public API (APIv2).

New features

APIv2's improvements include:

- A fully RESTful implementation, including routes with a more programmatic syntax.
- Datasets that are arranged in a logical hierarchy. Member datasets may be discovered by querying their parent node.
- Modalities, periodicities, and facets, which vary between datasets, that may be specified and queried programmatically.
- Customizable API returns: specify the columns, date range, and facets you wish to receive, thereby removing the need for client-side processing.
- Enhanced metadata, specific to the dataset.
- Responses to parameters both in the URL and in the HTTP header.

APIv1 availability

EIA will maintain support for its legacy API (APIv1) until at least January 2023. Some of APIv1's features are *deprecated*, meaning that they are available in the API's original data series and interface, but they are not in APIv2.

Table 1 summarizes the differences between the API versions.

Table 1. APIv2 changes

Feature	APIv1 (deprecated; only exists in APIv1)	APIv2 Equivalent
HTTP protocols	http and https (http to end soon)	Only https
Geoset query	Separate query with the *command* parameter	*Facet* property with values such as [stateid]
Relation and category	Separate queries with the *command* parameter	Relation and category are inherent to the RESTful route or are expressed in metadata if not in a tree hierarchy
Sort	N/A	A *sort* parameter
Pagination	*Start*, *end*, and *num* parameters in an HTTP header	*Offset* and *length* parameters
Update query	Quota-free query that returns all series in category and their last update expressed in datetime	Not supported

Overall Concepts

Examples in this guide

We will gradually build our API queries to harness the power of our API. To keep things simple, we will remain with a single-use case: the retail sales of electricity in Colorado. However, this command syntax applies to all of our APIv2 datasets.

In this document, italicized code represents deprecated v1 methods that are interacting with our API.

```
http://deprecated_code/
```

APIv2 call examples are shown in a bolded, fixed-width font:

```
https://api.eia.gov/API_route
```

API returns or HTTP headers are shown in a bordered call-out:

```
response: {
  Return from our API
}
```

In these examples, we've added whitespaces and carriage returns for clarity.

Many of our examples return very large data sets. For brevity, when we omit lines of a return, we'll denote them with ellipsis (…) as follows:

```
response: {
  Thing 1
  Thing 2
  Thing 3
  …
  }
```

We will use **highlighted, bold, shaded text** to indicate important code and output.

Using an API key

To call our API, you must use the unique API key assigned to you. Visit our Open Data pages (https://www.eia.gov/opendata/) to register for this free key. EIA will send the key automatically to the email address provided.

To use an API key, place it as a parameter after the route.

https://api.eia.gov/API_route?api_key=xxxxxxx

Replace **xxxxxx** with the API key transmitted in your email. You must use the API key.

Submitting requests to our RESTful API and legacy APIv1 Series ID

Our API accepts RESTful URL requests. For a primer on what REST is, and for general guides to RESTful APIs, consult the following resources:

- http://restfulapi.net
- http://en.wikipedia.org/wiki/Representational_state_transfer

In APIv2 and beyond, our data series are now organized in a tree-like hierarchy. Our program requests data by expressing a URL path through this tree, which looks like directories in a URL. This result is called a *route*. *For example, here's the route for retail sales of electricity (note, we always must use our individual API user's key):*

https://api.eia.gov/v2/electricity/retail-sales?api_key=xxxxxx

In APIv1, we had to know the series ID and express it in an URL call as follows:

http://api.eia.gov/~~series/?series_id=sssssss~~&api_key=xxxxxx

For the Colorado data mentioned before, the older v1 requesting call would have been:

http://api.eia.gov/~~series/?series_id=ELEC.SALES.CO-RES.A~~&api_key=xxxxxx

This APIv1 method is now deprecated . However, if you are fond of APIv1 Series IDs, you may invoke them using the special /series/ route, as follows. Note that we use a full route, but at the end we include the APIv1 Series_ID:

https://api.eia.gov/v2/seriesid/ELEC.SALES.CO-RES.A?api_key=xxxxxx

Supporting multiple versions of our API

The first node in the route is the version of the API we wish to use. This document invokes our version 2 of the API, so /v2/ appears in all requests.

```
https://api.eia.gov/v2/API_route?api_key=xxxxxx
```

Generating an API call

Before we return actual data from the API, we'll review the API's syntax and how to discover all the data available.

Choosing between parameter locations

Our API looks for parameters in two locations, either in the URL or in the HTTP headers that we send in the GET request.

GET places all of of our request variables in the URL itself. For example:

```
http://api.eia.gov/v2/electricity/retail-sales/data/?api_key=xxxxx
x&facets[stateid][]=CO&facets[sectorid][]=RES&frequency=monthly
```

In this document, we have placed all our API query parameters in the URL. Some systems enforce a maximum length for URLs, so the API also accepts parameters inside the HTTP GET request.

To submit this same API call with parameters in this manner, we place our parameters inside the HTTP headers we submit to our API server. Note that our API Key must always appear in the URL; it will not be detected in the HTTP headers.

```
https://api.eia.gov/v2/electricity/retail-
sales/data?Api_key=xxxxxx
 (standard HTTP headers here)
…
Content-Type: application/x-www-form-urlencoded
facets[stateid][]=CO
facets[sectorid][]=RES&frequency=monthly
```

Understanding returned error codes

If our query omits a required parameter or contains other syntax errors, the API will respond with the relevant HTTP error codes and, in some cases, additional JSON text explaining the error. For example, if we have data series by month, quarter, and year and we ask for millennia, the API will respond:

```
{
  error: "Invalid frequency 'millenially' provided. The only
valid frequencies are 'monthly', 'quarterly', and 'annual'.",
  code: 400
}
```

Warning messages

Similarly, if the API is able to return a data response to our request but is behaving in a potentially unexpected manner, it will include a warning in addition to the data return.

For example, our API only returns a maximum of 5,000 rows, even if more data points are responsive to your request. The API will return as many rows as it can, along with this warning:

```
{
  warning: "parameter out of range"
  description: "The API can only return 5000 rows in JSON format.
Please consider constraining your request with facet, start, or
end, or using offset to paginate results."
}
```

Debugging information

At the end of every return, the API will echo back what it understood to be our request. This response may be helpful while debugging advanced queries. The API will also identify the version of the API we are interacting with, so we can pair it with the correct technical documentation and other dependencies.

```
{
  response: {
    …
    },
  request: {
    command: "/v2/electricity/",
    params: {
      api_key: "xxxxxx"
      …
      }
    }
  apiVersion: "2.1.0"
}
```

Iterating through the API's tree and its metadata

Discovering datasets should be much easier in APIv2 because the API now self-documents and organizes itself in a data hierarchy. Parent datasets have child datasets, which may have children of their own, and so on. To investigate what datasets are available, we request a parent node. The API will respond with the child datasets (routes) for the path we've requested.

In our ongoing example, we're looking for retail sales of electricity in Colorado. Let's start at the top and ask about electricity data, as follows:

`https://api.eia.gov/v2/electricity?api_key=xxxxxx`

If we don't include **`/data`** as the last node in our route or if we request a parent node with multiple data series as children, the API will return metadata pertaining to our request. Some fields returned (depending on the series) may include:

- A long-form description
- Available frequencies and periodicities
- Optional data facets (such as a state location or PADD region)
- Child data series

```
response: {
  id: "electricity",
  name: "Electricity",
  description: "EIA electricity survey data",
  routes: [
    {
      id: "retail-sales",
      name: "Electricity Sales to Ultimate Customers",
      description: "Electricity sales to ultimate customer
by state and sector (number of customers, average price,
revenue, and megawatthours of sales).  Sources: Forms EIA-826,
EIA-861, EIA-861M"
    },
    {
      id: "electricity-power-operational-data",
      name: "Electric Power Operations (Annual and Monthly)",
      description: "Monthly and annual electric power operations
by state, sector, and energy source. Source: Form EIA-923"
    },
    {
      id: "rto",
      …
    },
    {
      id: "state-electricity-profiles",
      …
    },
    {
      id: "operating-generator-capacity",
      …
    },
    {
      id: "facility-fuel",
      …
    }
  ]
},
…
```

In this case, the **`/electricity`** route has several child routes. No data is specific to the master category of electricity. To explore one step further down our data hierarchy, we added our choice of these nodes to our previous URL routes, like so:

`https://api.eia.gov/v2/electricity/retail-sales?api_key=xxxxxx`

`https://api.eia.gov/v2/electricity/electricity-power-operational-data?api_key=xxxxxx`

```
https://api.eia.gov/v2/electricity/rto?api_key=xxxxxx

https://api.eia.gov/v2/electricity/state-electricity-profiles?api_key=xxxxxx

https://api.eia.gov/v2/electricity/operating-generator-capacity?api_key=xxxxxx

https://api.eia.gov/v2/electricity/facility-fuel?api_key=xxxxxx
```

Using this mechanic, we can programmatically iterate through our data hierarchy.

A word on API key limits

Some programmers may create recursion routines to scrape our 2M+ data series automatically. Although they are free to do so, these programs must throttle the number of the requests they make per second and per hour. If you exceed these tolerances, your API key will be automatically but temporarily suspended, and it will be automatically reactivated after a brief cool down period. You will receive an error message informing you if this condition occurs. More specific information about these throttles are documented on our Open Data web pages.

Examining a metadata request

Let's explore one of these paths. We'll search for retail sales of electricity. We're not using `/data` yet, so we wouldn't receive data values. However, we will receive a metadata return. Here's our API call:

```
https://api.eia.gov/v2/electricity/retail-sales?api_key=xxxxxx
```

```
response: {
  id: "retail-sales",
  name: "Electricity Sales to Ultimate Customers",
  description: "Electricity sales to ultimate customer by state
and sector (number of customers, average price, revenue, and
megawatthours of sales).
Sources: Forms EIA-826, EIA-861, EIA-861M",
  frequency: [
    {
      id: "monthly",
      description: "one data point for each month.",
      query: "M",
      format: "YYYY-MM"
    },
    {
      id: "quarterly",
      …
    },
      id: "yearly",
      …
    }

  ],
    facets: [
      {
        id: "stateid",
        description: "State / Census Region"
      }
      {
        id: "sectorid",
        //metadata for this facet "sector"
        …
      }
    ],
    data: {
      revenue: {
        alias: "Revenue from Sales to Ultimate Customers",
        units: "million dollars"
      },
      sales: {
        alias: "Megawatthours Sold to Utlimate Customers",
        units: "million kilowatthours"
      },
      price: {
        alias: "Average Price of Electricity to Utlimate
Customers",
        units: "cents per kilowatthour"
      },
      customers: {
        alias: "Number of Ultimate Customers",
        units: "Number of customers"
      }
    },
    startPeriod: 2001-01,
    endPeriod: 2022-07,
    defaultDateFormat: "YYYY-MM",
    defaultFrequency: "frequency"
    // Additional metadata here, if available
  }
```

From this return, we see that the following data dimensions are available:

- Three periodicities (frequencies) of data: monthly, quarterly, and annual
- Two facets for further filtering: location and sector
- Data columns that will contain individual values: revenue, sales, price, and customers
- Additional metadata such as the range of available data, and default parameters should we choose not to explicitly stipulate them

This example return contains the metadata specific to retail sales of electricity. Other routes, many concerning different energy sources, are likely to have different metadata, specific and germane to their context.

Facets and their available values

Determine the variables we can pass into the API to customize results

In the previous example, we asked for metadata about retail sales of electricity. We saw two facets: location and sector. To determine what the appropriate values for those are, we query on that facet itself Let's try asking for all available sectors by specifying the **`sectorid`** facet:

```
    https://api.eia.gov/v2/electricity/retail-sales/facet/sectorid/?api_key=xxx
    xxx
-----
```

```
response: {
  totalFacets: 6,
  facets: [
    {
      id: "COM",
      name: "commercial", alias: "(COM) commercial"
      },
      {
      id: "RES",
      name: "residential",
      alias: "(RES) residential"
      },
      {
      id: "ALL",
      name: "all sectors",
      alias: "(ALL) all sectors"
      },
      {
      id: "OTH",
      name: "other",
      alias: "(OTH) other"
      },
      {
      id: "TRA",
      name: "transportation",
      alias: "(TRA) transportation"
      },
      {
      id: "IND",
      name: "industrial",
      alias: "(IND) industrial"
      }
    ]
  },
…
```

The API reports six available values for the facet **`sectorid`**: COM, RES, ALL, OTH, TRA, and IND.

We can query on a **`sectorid`** not in this list, for example *xxx*. The API won't return an error—because that's a valid query. However, we won't receive any data returns either because the *xxx* sector doesn't have any data points.

Returning metadata versus specific data values

To request data points from the API, we stipulate **`/data`** as the final node in our API call. For example:

`https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx`

The API will now return all data columns relevant to our query.

```
response: {
    total: 7440,
    dateFormat: "YYYY",
    frequency: "annual",
    data: [
      {
        period: "2001",
        stateid: "AL",
        stateDescription: "Alabama",
        sectorid: "ALL",
        sectorName: "all sectors"
      },
      {
        period: "2001",
        stateid: "AL",
        stateDescription: "Alabama",
        sectorid: "CO",
        sectorName: "commercial"
      },
      …
    ]
  },
…
```

Note that there are no values here, yet. Later, we'll specify the data points we want to receive using the **`data[]`** parameter, which returns columns of data.

Parameters

Now that we have the API answering our requests and we know how to explore routes, metadata, and facets, let's get some data points out of it.

We may use a variety of parameters in our query to customize the return's results. We'll go over each parameter in turn. By doing so, we'll slowly build a robust API request.

Data []

(Optional, but required to receive data values) For the given route, specifies the data columns available to be returned

To retrieve data points and their values from the API, we need to specify the specific columns we are interested in. In this document, we've been asking about electricity residential sales, but many data points about that subject matter are available. As of early 2022, our API has data values on revenue, sales, price, and number of customers.

In earlier examples, when we asked about the metadata, the API responded with these available data points:

`https://api.eia.gov/v2/electricity/retail-sales/?api_key=xxxxxx`

```
…
data: {
  revenue: {
    units: "dollars"
    },
  sales: {
    units: "kilowatthours"
    },
  price: {
    units: "dollars per kilowatthour"
    }
  //Additional data values, if available
  }
```

Given these columns, let's ask for the price. Remember, in addition to specifying the column in the **`data[]`** parameter, we must also specify **`/data`** as the last node in the route:

> **https://api.eia.gov/v2/electricity/retail-sales/data/?api_key=XXXXXX&data[]=price**

The response is a very large data set. We didn't specify any facets or filters, so the API returned as many values as it could. **The API will not return more than 5,000 rows of data points**. However, it will identify the total number of rows that are responsive to our request in the response header. In this case, 7,440 data rows match the API request we just made.

Here are the first few rows (as data is added and updated, these data points may change):

```
response: {
  total: 7440,
  dateFormat: "YYYY",
  frequency: "annual",
  data: [
    {
      period: "2010",
      stateid: "AZ",
      stateDescription: "Arizona",
      sectorid: "TRA",
      sectorName: "transportation",
      price: 0,
      price-units: "cents per kilowatthour"
    },
    {
      period: "2010",
      stateid: "AR",
      stateDescription: "Arkansas",
      sectorid: "ALL",
      sectorName: "all sectors", price: 7.28,
      price-units: "cents per kilowatthour"
    },
    … //additional returns
    ]
  }
```

We can request multiple data columns be returned. Let's add the revenue column to our last query:

> **https://api.eia.gov/v2/electricity/retail-sales/data/?api_key=XXXXXX&data[0]=price&data[1]=revenue**

```
data:
  [
    {
      period: 2010,
      stateid: "AZ",
      stateDescription: "Arizona",
      sectorid: "TRA",
      sectorName: "transportation",
      price: 0,
      revenue: 0,
      price-units: "cents per kilowatthour",
      revenue-units: "million dollars"
    },
    …
  ]
```

We do not need to manually stipulate the members of the array; our API will automatically create it for us. This query also works:

```
https://api.eia.gov/v2/electricity/retail-sales/data/?api_key=XXXXXX
&data[]=price&data[]=revenue
```

and will produce the same return:

```
data:
  [
    {
      period: 2010,
      stateid: "AZ",
      stateDescription: "Arizona",
      sectorid: "TRA",
      sectorName: "transportation",
      price: 0,
      revenue: 0,
      price-units: "cents per kilowatthour",
      revenue-units: "million dollars"
    },
    …
  ]
```

Facets

(Optional) Filters the API's response based on our requested location, sector, or other data filters

Most of our data series have one or more query facets. If we query a series that has these facets without stipulating one or more of them, the API will respond with all matching data. The result can be a very large return. Facets enable us to filter the data of concern to us, shrinking the size of the returns to a more manageable size.

For example, our retail sales of electricity has the **`location`** and **`sector`** facets. If we query the route (without specifying **`/data`**), the API will tell us the facets that are relevant to that route.

```
https://api.eia.gov/v2/electricity/retail-sales/?api_key=xxxxxx
```

```
…
  facets:
    [
      {
        id: "stateid",
        description: "State / Census Region"
      },
      {
        id: "sectorid", description: "Sector"
        description: "sector"
      }
    ]
```

Let's go back to our data query above and its very large return. To focus on residential sales, we'll add the **`sectorid`** facet, and set it equal to **`RES`** (**`RES`** is the **`sectorid`** for residential, as we learned from our metadata queries earlier).

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=
price&facets[sectorid][]=RES
```

Now, we receive data that's only for the residential sector.

We may stipulate more than one facet in a call. Do so by invoking multiple facet parameters. To whittle down our query, let's specify electricity prices in Colorado *and* those for the residential sector.

To do so, we'll add the **`facet[stateid]`** and set it to **`CO`**. Remember to ask for a column return, in this case, price:

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[
]=price&facets[sectorid][]=RES&facets[stateid][]=CO
```

And then the API returns only price data for the residential sector in Colorado:

```
response: {
  total: 20,
  dateFormat: "YYYY",
  frequency: "yearly",
  data: [
    {
      period: 2001,
      stateid: "CO",
      stateDescription: "Colorado",
      sectorid: "RES",
      sectorName: "residential",
      price: 7.47,
      price-units: "cents per kilowatthour"
    },
    {
      period: 2002,
      stateid: "CO",
      stateDescription: "Colorado",
      sectorid: "RES",
      sectorName: "residential",
      price: 7.37,
      price-units: "cents per kilowatthour"
    },
    …
  ]
}
…
```

Frequency

(Optional) Stipulates the periodicity of the data, if multiple options exist

Many of our data series are assembled in different periodicities – how often an event occurs or is measured. For example, we may have data grouped annually, quarterly, monthly, or even daily. The frequency query parameter stipulates the periodicity we want. Here, we'll ask for residential prices, tabulated monthly.

`https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly`

Note that the period parameters have changed, indicating that we are now viewing monthly data:

```
response: {
  total: 251,
  dateFormat: "YYYY-MM",
  frequency: "monthly",
  data: [
    {
      period: "2001-01",
      stateid: "CO",
      stateDescription: "Colorado",
      sectorid: "RES",
      sectorName: "residential",
      price: 6.71,
      price-units: "cents per kilowatthour"
    },
    …
```

If we omit this parameter, the API will respond with the default periodicity for that series. We can confirm what periodicity we're looking at by looking at the frequency metadata entry.

Date range

(Optional) Stipulate a start and end date restriction for the data

We can stipulate that the API only return data that lies within a specific date range. To do so, we add these optional parameters to our query, where *yyyy* is the year, *mm* is the month, and *dd* is the day.

Using our ongoing example, let's stipulate we only want to see February and March 2008 residential retail-sales of electricity in Colorado. The start parameter instructs the API only to return data after a specific point, and the end parameter stipulates the latest date. Using both of them in the same call, we can request data constrained to a specific time span:

Start date

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly&start=2008-01-31
```

End date

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly&end=2008-03-01
```

Start and end date together

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly&start=2008-01-31&end=2008-03-01
```

We receive two rows in return, the data points for February 2008, and March 2008.

In the last example, we specified a start date on January 31. Remember, the first part of this return stipulated that the series was monthly and the format of dates was YYYY-MM:

```
response: {
  total: 2,
  dateFormat: "YYYY-MM",
  frequency: "monthly",
}
```

The datestamp of the February monthly data point, **`2008-02`**, mathematically occurs before **`2008-02-01`**. If we were to stipulate **`&start=2008-02-01,`** or February 1, 2008, we wouldn't receive February's data. That's why we stipulate the day before 2008-02, or 2008-01-31 (January 31st).

Sort results

(Optional) Orders the data by the column and in the direction we stipulate

We can request that the results of our query be ordered by any column, or multiple columns, in ascending or descending order.

We invoke the **`sort`** array with this format:

```
[SORT_PRECEDENCE_HERE][column]=COLUMN_NAME_HERE
```

starting with a **`sort_precedence`** of 0.

We can stipulate the element **`[direction]`** as either **`asc`** or **`desc`** to dictate the sort direction. For example, to order our results so that the most-recent data is returned first, we sort on the **`period`** column in descending order:

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly&sort[0][column]=period&sort[0][direction]=desc
```

This code changes the order of data points appropriately:

```
response: {
  ..
  [
    {
      …
      period: "2021-11,"
    },
    {
      …
      period: "2021-10,"
    },
    {
      …
      period: "2021-09,"
    },
…
  ]
…
}
```

If we do not specify an order, the API will use the default sort order for that data series.

Pagination

(Optional) Returns a subset of eligible rows responsive to the query

Our request may generate more rows than we'd like to ingest in one API call. EIA's API limits its data returns to the first 5,000 rows responsive to the request (300 rows if we request XML format).

We can override this behavior by using the length parameter. Using our most recent example of sorting by recent data first, let's imagine we only want the most recent year of data. Because we're stipulating a monthly frequency, we'd only want 12 rows of data. We assign the `length` parameter a value of 12:

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly&sort[0][column]=period&sort[0][direction]=desc&length=12
```

We receive the first 12 rows of data the API can find, based on any sort or facet commands we give it.

`Offset` stipulates the row number the API should begin its return with, out of all the eligible rows our query would otherwise provide.

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly&sort[0][column]=period&sort[0][direction]=desc&offset=24
```

In the above example, the API will skip over the first 24 eligible rows (**`offset=24`**), which translates into 24 months (**`frequency=monthly`**).

We can combine these two parameters to page through our data. To skip two years (**`offset=24`**) and request a year of returns (**`length=12`**), we'd write:

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&frequency=monthly&sort[0][column]=period&sort[0][direction]=desc&offset=24&length=12
```

No matter what we stipulate with these two parameters, our API will always return the total number of rows otherwise responsive to our request. Even if we only request 12 rows, as above, the data return informs us of the total rows available:

```
response:
  {
    total: 251,
    dateFormat: "YYYY-MM",
    frequency: "monthly",
…
}
```

Output format

(Optional) Specifies XML or JSON output.

By default, our API will provide data returns in JSON format. However, you may specifically request XML output with the **`out`** parameter:

```
https://api.eia.gov/v2/electricity/retail-sales/data?api_key=xxxxxx&data[]=price&facets[sectorid][]=RES&facets[stateid][]=CO&out=xml
```

```
<eia_api>
  <response>
  …
    <row>
      <period>2009</period>
      <stateid>CO</stateid>
      <stateDescription>Colorado</stateDescription>
      <sectorid>RES</sectorid>
      <sectorName>residential</sectorName>
      <price>10</price>
      <price-units>cents per kilowatthour</price-units>
    </row>
  </response>
  …
</eia_api>
```

For performance reasons, the API can only return a ***maximum of 300 rows*** when producing XML output. You may use the pagination features as described above to parse through the entire data return. You will see a warning embedded in the XML output to alert you of this limitation.

You may also explicitly stipulate JSON output (`out=JSON`), however you don’t need to make this stipulation because omitting the parameter defaults the API to JSON format.

--- END ---